A mathematical model of NIAM conceptual models

A thesis submitted to the University of Manchester Institute of Science and Technology for the degree of Doctor of Philosophy

John G Harris

Department of Computation October 1998

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Acknowledgements

I thank my supervisor Thierry Scheurer and the students and staff of UMIST for making the last few years both interesting and enjoyable.

Abstract

Anyone given the job of designing a product must choose a suitable design technique and must then use it correctly. If design tools are to help them then the tools should ensure that the technique is used correctly without restricting proper use. NIAM is a conceptual data modelling technique for designing databases. A simple test to decide when NIAM can and cannot be used would benefit product designers. Simple rules to say when the construction and alteration of a data model are proper would benefit both product designers and design tool designers. As far as can be ascertained this information has not appeared in the literature. Therefore the purpose of this work is to answer some specific questions with the general theme

"When can I use NIAM and how might design tools help me?".

We start by demonstrating the diversity of objects that can rightfully be called databases. We do not restrict ourselves to computer systems. We then develop a new set-theoretical model of NIAM conceptual data models. The model is defined in Scheurer's Feature Notation and all results are proved rigorously. We prove that each data model we have modelled is well-formed in that it provides a well defined specification of a database. We also prove that we have modelled all possible well-formed NIAM data models, except those requiring the database to have unconventional mathematical properties. (Ordinary commercial and industrial databases are conventional). We use these results to devise a simple test to decide when NIAM can be used and when it cannot.

We define several editing operations on data models. Some do incremental changes; the others do more general "cut and paste" changes. We state simple preconditions for each operation and prove that if the preconditions are obeyed then the result of operating on a well-formed data model is also well-formed. We show that any editing action likely to be needed can be composed from these operations. It would be a straightforward matter to implement these operations and their precondition tests in a design tool. We also define a new class of equivalent constructions in NIAM data models. We devise and prove a simple test for equivalence.

0.1 Contents

0.2 Symbols

0.2.1 Logic

$\neg$	Not
$\vee$	Or
$\wedge$	And
$\Rightarrow$	Implies; If … then …
$\Leftrightarrow$	Equivalent; If and only if
$\forall$	For all
$\exists$	There exists
$\exists 1$	There exists exactly one
$\bullet$	Punctuation separating a quantified variable from its associated Wff

0.2.2 Sets

$\in$	Member of
$\notin$	Not member of
$\varnothing$	The empty set
$\subseteq$	Subset
$\subset$	Strict subset
$\cup$	Union (of two sets)
$\bigcup$	Union (of the sets in a set)
$\cap$	Intersection (of two sets)
$\bigcap$	Intersection (of the sets in a set)
$\backslash$	Relative complement
$+$	$\cup$, where $x + y$ emphasises that x and y are disjoint
$-$	$\backslash$, where $x - y$ emphasises that $y \subseteq x$
$\{\ \}$	Set. E.g $\{1, 3\}$
$_d$	By definition. E.g $x =_d y$
:	Is a member of by definition. E.g $x : Y$ Also, in a set definition emphasises that x is not a parameter, alias not a free variable. E.g $\{ \bigcup x \mid x : Y \}$
$\langle\ \rangle$	Primitive couple, alias ordered pair. E.g $\langle 1, 3\rangle$ $\langle x, y\rangle =_d \{\{x\}, \{x, y\}\}$
$\leftrightarrow$	The class of all relations with the given Domain and Codomain
$+\!\!\rightarrow$	The class of all (possibly) partial functions with the given Domain and Codomain
$\rightarrow$	The class of all total functions with the given Domain and Codomain
$\mapsto$	Maplet; individual couple in the graph of a relation or function
$\langle$	Domain restriction of a binary relation or function. E.g $X \langle R$

〈**.**	Codomain restriction of a binary relation or function. E.g X 〈**.** R
◊	Both Domain and Codomain restriction of a homogeneous binary relation or function. E.g $X \lozenge R$
∘	Function composition. E.g $F{\circ}G$
$(f \mid x \mapsto y)$	Function override. For any possibly partial function f, if $g = (f \mid x \mapsto y)$ then $g(x) = y$, otherwise g is the same as f (note that f is not necessarily defined at x)
[[ ]]	Image operator. E.g F[[X]], the image of X under the function F
[ ]	1) Sequence. E.g [a, e, a, o] 2) Consecutive natural numbers. E.g [m..n], the set of natural numbers from m to n inclusive
( )	1) n-tuple. E.g (1, 3, 7) 2) Family. E.g $(f_i \mid i : I)$, the total function f whose domain is the index set I 3) Distributed function. E.g $(f_i : D_i \mid i : I)$, the family whose elements are constrained by the domain function $D =_{\mathrm{d}} (D_i \mid i : I)$ so that $\forall\ i : I \bullet f_i \in D_i$
ϕ	The 0-tuple, alias ( ), see Section 4.3.2
Φ	The nullary cartesian product, { ϕ }, see Section 4.3.2

0.2.3 Miscellaneous

Const	Constants are written in sans-serif characters
Var	Variables are written in italic characters
.:	Feature Notation punctuation, introducing a fixed feature; see Section 3.3
□	The end of a proof

Graphic symbols used in conceptual data modelling are given in the literature survey; see Sections 2.1.1, 2.1.3, 2.2.2, and 2,2,3.

0.3 Terms

0.3.1 General acronyms

CAD	Computer Aided Design
CASE	Computer Aided Software (or System) Engineering
DB	Database
DBMS	Database Management System
EAR	Entity Attribute Relationship (diagram)
ER	Entity Relationship (diagram)
FOL	First Order Language (or Logic)
FORM	Formal Object Role Modelling, a variant of NIAM; see Section 2.1.1.1
iff	If and only if
NIAM	Nijssen's Information Analysis Methodology (originally) Natural-language Information Analysis Method (more recently) (fact oriented conceptual data modelling)
OO	Object-Oriented (programming language or database)
ONF	Optimal Normal Form, a NIAM term; see Section 2.1.1
ORM	Object Role Modelling, a generalisation of NIAM; see Section 2.1.1.1
RDB	Relational database
UMIST	The University of Manchester Institute of Science and Technology
VDM	A specification notation
VNB	Von Neumann Bernays set theory (as in Hamilton [1982], p145-156)
w.r.t	With respect to
Z	A specification notation
ZF	Zermelo Fraenkel set theory (as in Enderton [1977])
Wff	Well Formed Formula

0.3.2 General terms

Atomic operation	A NIAM term; see Section 2.1.1
Attribute	A modelling object used in Chen-style conceptual data models; see Section 2.2.2
Cod	Codomain of a binary relation or function
Conceptual data model	A database design term; see Sections 2.1.1, 3.2
Constraint	A database term; see Section 2.1.1, 3.2.2, 7.3
Def	Definition domain of a binary relation or function
Definitional system	See Section 2.3.3.2

Derived Fact Type	A NIAM term; see Section 2.1.1, 7.3.3
Distributed function	Family with restricted values; see under "( )" in Symbols, above
Dom	Domain of a binary relation or function
Dynamic constraint	A database term; see Section 2.1.1, 3.2.2, 7.3
Elementary fact	A NIAM term; see Section 2.1.1
Entity	A NIAM term; see Section 2.1.1, 4.1.1
Entity Type	A NIAM term; see Section 2.1.1, 4.1.1
ER diagram	A database design term; see Section 2.1.1
Fact	A NIAM term; see Section 2.1.1, 4.1.1
Fact Type	A NIAM term; see Section 2.1.1, 3.2.2, 4.1.1
Feature Notation	Notation for defining set-theoretical models; see Scheurer [1994] and Section 3.3
Flattening	Transforming a tuple into an equivalent tuple whose values are elementary; see Section 5.5.1
Gr	Graph of a relation or function
Information diagram	A NIAM term; see Section 2.1.1
Inherit	A NIAM usage; see Section 2.1.1
Instance (of a database)	A database term; see Section 2.1.1, 3.1.1, 7.2, 7.4
Label	A NIAM term; see Section 2.1.1, 4.1.1
Label Type	A NIAM term; see Section 2.1.1, 4.1.1, 7.1
Lexical object	A NIAM term; see Section 2.1.1
Mandatory (role constraint)	A NIAM term; see Section 2.1.1
Mapping oriented approach	See Section 2.1.2
Meta-schema	A NIAM term; see Section 2.1.1, 4.5.2
Nat	The set of natural numbers, 0, 1, 2, …
Nested	A NIAM term; see Section 2.1.1
NIAM-style	NIAM or a similar technique using the same principles as NIAM
Num	Number of members of a finite set (its cardinality). E.g $\mathsf{Num}(\{a, b\}) = 2$
Object Type	A NIAM term; see Section 2.1.1
Objectified	A NIAM term; see Section 2.1.1
Optional (role constraint)	A NIAM term; see Section 2.1.1
Opp	Opposite (of a relation)
Plays	A NIAM usage; see Section 2.1.1

Pop, Population	A NIAM term; see Section 2.1.1, 7.2
Pow	Power set of a set
Predicator	Objects used in the Predicator Model, q.v.
Predicator Model	A model of NIAM data models; see Section 2.1.2
Ran	Range of a binary relation or function
Relation	As in mathematics; not necessarily a table in a relational database
Role	A NIAM term; see Section 2.1.1, 4.1.1
Schema transformation	A NIAM term; see Section 2.1.1, 5.5
Static constraint	A database term; see Section 2.1.1, 3.2.2, 7.3
Subtype, Subtype constraint	A NIAM term; see Section 2.1.1, 7.3.4
Total role constraint	A NIAM term; see Section 2.1.1, 7.3.5
Transaction	A NIAM term; see Section 2.1.1, 7.4
Tuple	As in mathematics, but usually represented as a family; see under "( )" in Symbols above and Section 4.1.1, 4.3
Tuple oriented approach	See Section 2.1.2
Uniqueness constraint	A NIAM term; see Section 2.1.1, 7.3.5
Value	More recent name for a label, see above
Value Type	More recent name for a Label Type, see above

0.3.3 Special terms

These are terms with possibly new meanings. Their definitions will be found in the quoted sections.

Actual database	3.1.2
Ancestor	4.3.1
Base conversion	5.4.1
Base domain	6.2
Basic domain	6.2
Conceptual database	3.1.2
Completion sequence	4.5.4
Construction sequence	4.3.1
Core data model	3.2.2, 4
Database	2.1.1, 3.1.1, 7.4
Descendant	4.3.1
Domain function	4.3.1

0.3.4 Special classes, sets, and operators

These are the names of the more prominent classes, sets, and operators that are introduced in this work. Their definitions will be found in the quoted sections. The list includes feature names. These are written in italics with a leading underline. Thus *PObjs*, *DObjs*, *D'Objs*, etc, are listed as *_Objs* and listed as though they start with the letter "O".

_Carts	Primary feature of any member of SetCart	6.3
_Chos	Secondary feature of any member of SetCartFin	6.3
CompSeq	Operation on DaMod0 returning a completion sequence	4.5.4
_Conn	Primary feature of any member of PreMod	4.2.3
ConsSeq	Operation on DaMod0 returning a construction sequence	4.5.4
ConsSeqFin	Operation on SetCartFin returning a construction sequence	6.3
_ConSpec	Secondary feature of any member of DaMod1	7.5
DaMod0	Subset of PreMod : model of all core conceptual data models	4.2.3
DaMod1	Extended DaMod0 : model of all conceptual data models	7.5
_Def	Definition domain of a relation or (possibly partial) function	General
_Descs	Secondary feature of any member of DaMod0	4.4.2
Diff	Editing function on PreMod, hence on DaMod0	5.2.2
_Doms	Secondary feature of any member of SetCart	6.3
_DomsBase	Secondary feature of any member of SetCart	6.3
_DomsBasic	Secondary feature of any member of SetCartNom	6.3
_DomsNom	Secondary feature of any member of SetCartNom	6.3
EmpDaMod0	Empty model; a generator of DaMod0	4.2.3
_EnSets	Secondary feature of any member of PreMod	4.3.2
Entities	Set of all possible entities used in conceptual data modelling	4.2.3
_Entities	Secondary feature of any member of PreMod	4.3.2
Facts	Set of all database tuples defined by DaMod0	4.3.2
_Facts	Secondary feature of any member of DaMod0	4.3.2
_FlatArity	Secondary feature of any member of DaMod0	5.5.3
_FlatCart	Secondary feature of any member of DaMod0	5.5.3
_FlatDomf	Secondary feature of any member of DaMod0	5.5.3
FlatEq	Equivalence relation on DaMod0 : same flattened tuples	5.5.3
_FlatIndex	Secondary feature of any member of DaMod0	5.5.3
_Flatten	Secondary feature of any member of DaMod0	5.5.3
_Gr	Graph of a relation or function	General
_Indexes	Secondary feature of any member of SetCart	6.3
_InDom	Secondary feature of any member of SetCartDis	6.3
_IsDomOf	Secondary feature of any member of SetCart	6.3
_IsDomOfNom	Secondary feature of any member of SetCartNom	6.3
ListV	Class of all valued lists	4.5.4
Merge	Editing function on PreMod, hence on DaMod0	5.2.2
Move	Editing function on PreMod, hence on DaMod0	5.2.2
_Noms	Primary feature of any member of SetCartNom	6.3
ObChos	Set of all choice functions on Objects	4.5.4
Objects	All subsets of Entities and all subsets of Roles	4.2.3
_Objs	Primary feature of any member of PreMod	4.2.3
_Pop	Secondary feature of any member of DaMod1	7.5

_Preds	Secondary feature of any member of DaMod0	4.3.2
PreMod	Precursor set, superset of DaMod0	4.2.3
_Ran	Range of a relation or function	General
_Rank	Secondary feature of any member of DaMod0	4.4.2
Roles	Set of all possible roles used in conceptual data modelling	4.2.3
_Roles	Secondary feature of any member of PreMod	4.3.2
_RoSets	Secondary feature of any member of PreMod	4.3.2
SetCart	Class of all sets of cartesian products	6.3
SetCartDis	Subclass of SetCart; disjoint index sets	6.3
SetCartFact	Subclass of SetCartFin; restricted to Roles and Entities	6.3
SetCartFin	Subclass of SetCartNom; finite index sets and domains	6.3
SetCartNom	Extension of SetCartDis; nominated indexes	6.3
StrucEq	Equivalence relation on DaMod0 : same type structure	5.3.2
_Succs	Secondary feature of any member of DaMod0	4.3.2
Tear	Editing function on PreMod, hence on DaMod0	5.2.2
Tuple	Class of all fact-style tuples	4.3.2
_TyStruc	Secondary feature of any member of DaMod0	5.3.2
TypeStruc	Subset of DaMod0 : all type structure descriptions	5.3.2
_Unflatten	Secondary feature of any member of DaMod0	5.5.3
_UsedBy	Secondary feature of any member of PreMod	4.3.2
_Uses	Secondary feature of any member of PreMod	4.3.2

1 Introduction

1.1 The problem

Suppose a bridge collapses, and suppose the design calculations say this cannot happen. What has gone wrong? Perhaps the calculations were done wrongly. Perhaps the wrong calculations were done. Perhaps the calculations do not apply to that kind of bridge. Perhaps the bridge was built incorrectly.

One of the problems facing a designer is to choose an appropriate design technique and then to use it properly. Designers assessing a candidate technique need to know two things. First, they need to know for which kind of system the technique is certainly appropriate and for which kind it is certainly not. Possibly there are systems for which it is uncertain. Second, they need to know which kind of system is to be designed. It is obviously desirable that they can characterise the system before time is wasted on an inappropriate technique. Having chosen a technique the designers must then restrict themselves to operations that are proper and avoid operations, such as division by zero, that are not. For this they need to know which are the proper operations in the chosen technique.

The choice becomes more difficult if the design techniques require an investment in training and computer aided design tools. Now the designers will expect to use the chosen technique for several systems. They must guess which kind of system is to be designed in the near future. They must make this guess in sufficient detail to choose a technique. In addition, they must choose design tools that implement desirable operations and avoid improper operations.

To complicate matters, there will be several different but overlapping design techniques that need to be chosen. The designers must choose a technique for the economic justification of the bridge, for traffic prediction, for structural integrity, for manpower and material quantities, and so on. Each technique must be chosen wisely. An economic "collapse" is just as undesirable as a mechanical collapse.

Bridges very rarely collapse nowadays but the same is certainly not true of computerised information systems. The common perception is that far too many information systems are unsatisfactory or worse. The design of bridges and information systems has in common the concurrent use of different design techniques, an investment in training and software tools, and the need to choose appropriate techniques and to use them properly. Information system designers need to know the capabilities of candidate design techniques and need to know their proper operations. They also need to know how to characterise information systems in order to choose appropriate techniques.

At the centre of most information systems there is a database that holds information to be acted upon by the rest of the system. Many different techniques for designing these databases have been published. Unfortunately, the publications, ranging from short papers to large textbooks, do little or nothing to help database designers decide which techniques are suitable for which kinds of database. They also say little or nothing about proper and improper operations. (That is, operations on **design** information. There is a vast literature on operations on databases themselves.)

Conceptual data modelling is one class of database design technique. A conceptual data model prescribes the permitted contents of the database and also describes the things, such as people, goods, and invoices, that the database refers to. A conceptual data model acts as a formal record of the meaning to be attributed to the contents of the database it prescribes (which, after all, is no more than an organised collection of bit-strings).

There are several families of conceptual data modelling technique. They differ in their basic assumptions or in the constructs they use. Within most families there are several variants, differing in minor details or in notation. One family stands out as being both coherent and inherently plausible. Its best-known variant is called NIAM. Although there is an excellent textbook, Nijssen and Halpin [1989], describing how to use NIAM it appears that nothing has been published about when the technique can and cannot be used. Database designers must read the book, understand it, and then imagine its use in their current projects. Nor is much said about operations on NIAM conceptual data models. If the designers alter a model or re-use part of it in another project then they must decide for themselves whether the result is sensible. The designers of computerised design tools must also decide for themselves which operations to permit. Moreover, they must convince their users that any forbidden operations would be improper.

NIAM is not the most popular database design technique at present yet it deserves to be considered by database designers. It is unsatisfactory that database designers and tool designers have no access to important information about its capabilities and use. Therefore, the objective of this work is to answer the following questions about the NIAM conceptual data modelling technique and its close relatives.

1 What, if anything, does a conceptual data model prescribe?

2 Can the rest of the development work on a database be classified as "implementation" : deciding *how* rather than *what*?

3 What simple test, if any, will recognise when NIAM can be used; and when it cannot?

4 What are the proper operations for constructing NIAM conceptual data models, for altering them, and for re-using parts in other projects?

5 Can computerised design tools facilitate all reasonable operations while precluding improper ones?

6 What are the essential components and structures that any computerised design tool must implement?

1.2 The plan

The objective is to answer the questions at the end of the previous section. This is done by constructing a new set-theoretical model of "all" NIAM conceptual data models, and then using it to obtain answers to the questions. In more detail :

Literature survey
Start with a condensed tutorial on the NIAM database design technique. Follow this with a description of two published mathematical models of NIAM conceptual data models. Neither is considered suitable for achieving the objective. Continue with brief descriptions of mathematical models of other database design techniques. Finish by mentioning other work that has been found to be useful here.

Context
To say what NIAM can and cannot do it is necessary to know what is and what is not a database. It is also necessary to understand what a conceptual data model purports to do. Determine the principles and essential characteristics of databases and of conceptual data models. Observe that any data model has a core structural part, and that the rest of the data model consists of values of various kinds attached to appropriate points in the structure. Observe also that data models are usually built incrementally, with each increment being a data model in its own right.

Core model : structure
Create a model of "all" NIAM conceptual data models, but only of the core structural part of each. The result is an inductively generated set : each member models a particular core structure; each generator mimics a natural elementary step in the incremental construction of a data model. Investigate the properties of this set. In particular, show that each member uniquely determines an essential database characteristic. Thus each member models the structure of a "proper" data model.

Core model : operations
Define some operations on core models. Show that the result of each operation will be a core model provided simple preconditions are satisfied. Show that these operations are complete : any core model can be transformed into any other core model by a suitable composition of operations. Thus a design editor can prevent the creation of data models whose structure is improper without restricting the designer's freedom.

Completeness model
Create a model of all conceivable cases of the essential database characteristic referred to earlier. Show that the cases conforming to some simple and reasonable restrictions are exactly those determined by some member of the core model. Conclude that the core model describes all useful core structures, and no others. Observe that the restrictions help to describe the capabilities of NIAM.

Core model extended
Extend the core model to include the remaining features of NIAM conceptual data models. As the extensions are mostly straightforward, even trivial, this need not be done in full detail.

Conclusion
Display the questions again and state the answers that have been obtained. State any

further results that are relevant to database design, to NIAM, and to design tools, and also any relevant to the art of using models built from sets. Indicate what further work could usefully be done in the future.

2 Literature Survey

This section surveys the literature relevant to the questions that were asked in Section 1.1. Much has been written about databases but most of it concerns the implementation and manipulation of databases themselves. The literature on database design techniques tends to be rather spasmodic and does not lend itself to a continuous historical treatment. Thus the survey devotes a separate sub-section to each group of closely related publications. Each sub-section consists of references followed by a summary of the publication(s), usually followed by comments on their strengths, weaknesses, and appropriateness.

The survey is split into three parts : major, minor, ancillary.

The major part, Section 2.1, starts with a summary of the main features of the NIAM conceptual data modelling technique. This is followed by summaries of two mathematical models of NIAM. (Only one other model has been published, as far as can be ascertained. It has been classified as minor.) The major part finishes with brief outlines of two very detailed mathematical models of other styles of conceptual data model.

The minor part, Section 2.2, briefly summarises publications concerning variant notations, other styles of conceptual data model, and other styles of mathematical model. It also includes some criticisms taken from a review of CASE tools (Section 2.2.2).

The ancillary part, Section 2.3, covers topics that can be regarded as useful background knowledge such as database theory and set theory. It also includes papers criticising the quality of definition of most database design techniques and the helpfulness of the tools supporting them (Section 2.3.3.4).

Where no other organisation seems appropriate the publications are introduced in approximate date order, earliest first.

2.1 Major

2.1.1 Nijssen's data modelling method (NIAM) - Leung & Nijssen [1988]; Nijssen and Halpin [1989]

References

[LN]	Leung, C M R and Nijssen, G M [1988]. Relational database design using the NIAM conceptual schema. Information Systems, Vol 13, No 2, p219-227.
[NH]	Nijssen, G M and Halpin, T A [1989]. Conceptual schema and relational database design : A fact oriented approach. Prentice-Hall of Australia Pty Ltd. *Library reference : Joule 001.6442/NIJ.*

Highlighting

Special terms defined in the paper and book are highlighted in bold at the point where they are introduced.

Summary

This paper and book describe a style of conceptual data modelling that is intended for use in the design of large corporate databases. The style is closely associated with G M Nijssen and is variously called **fact oriented data modelling** and **NIAM** (originally standing for Nijssen's Information Analysis Methodology).

The paper, [LN], is a summary of the principles and notation of NIAM, aimed at readers with some background knowledge of conceptual data modelling. The book, [NH], is a complete textbook on NIAM which assumes no prior specialised knowledge (though some examples use Australian terms such as "footie" for football).

There are many topics to be covered, so this summary starts with a list of page references and then describes the topics in a natural order with little reference to their origins.

Page references for the relevant topics in the [LN] paper are :

p219-221	The NIAM manifesto : a fifth generation design method for fourth generation implementations.
p220-222	The basic concepts and graphical notation.
p222-224	The algorithm for transforming a data model into a relational database design in what is called Optimal Normal Form (ONF).
p224-226	A prototype CAD tool for data modelling.

Page references for the relevant topics in the [NH] book are :

p9-23	Introduction to the principles and main concepts.
p29-65	Detailed description of the basic concepts and graphical notation.

p66-90	Detailed description of, and notation for, uniqueness constraints.
p109-136	Populations and population constraints, including subtypes.
p157-170	Referencing and identification schemes.
p216-244	Equivalent schemas (informal) and simple schema transformations.
p245-279	Relational database design, including the ONF algorithm.
p291-302	The conceptual meta-schema.
p303-314	Comparison of NIAM with other methods.

A description of the relevant topics now follows.

The purpose of a **database** is to hold an evolving set of **facts**, such as 'George smokes', 'Carol studies at UMIST', 'meet Alan on Friday at The Unicorn'. The purpose of a **conceptual data model** is to specify the facts that the database must be capable of holding, and to specify the combinations of facts that are to be regarded as legitimate. The database will usually hold its facts in an encoded form. For instance, facts about cars may be held indirectly in the form of facts about their registration marks. Although it is convenient to say "the" database, there may in practice be several databases conforming to the same specification and evolving independently.

Each **fact** specified in a conceptual data model is defined to be a **tuple** that is explicitly represented as a family. Each element of the family is constrained to belong to a specified domain. That is, each fact f is a distributed function $f =_{\mathrm{d}} (f_r : D_r \mid r : R)$ where the sets D_r are the domains of f. From now on D will be called the domain function of f. The members of the index set R, which must be finite and not empty, are called **roles**; standard usage is to say that " f_r **plays** the role r in the fact f ". For instance, we could say that Carol plays the role "studies" in the fact 'Carol studies at UMIST', whereas UMIST plays the role "studied at".

Each element f_r of the fact f is of one of two kinds. It can be a primitive object, such as a person or a registration mark, whose internal structure is not defined in the data model. Alternatively it is one of the facts, alias tuples, specified by the data model. This allows one fact to reference another, as in 'Carol studies Physics' referenced by 'This enrolment is worth 5 credits'.

Each primitive object is classified as either an entity, a label, or both. A **label**, alias **lexical** object, is one that can be represented directly in the database, such as a number or a character string. An **entity** is an object in the system being modelled that is of direct interest, such as a person or a car. Typically, labels are used only to identify entities, and are not of direct interest. However, it is possible for an object to be both an entity and a label. For instance, the car registration mark "G 123 XYZ" is a label which can identify a car, but in a database recording the purchase and sale of registration marks it is also an object of interest. The classification of an object as a label or an entity depends on the context (e.g can pictures be represented directly?) and can also be a design decision (e.g should pictures be held inside the database?).

This classification is significant when the conceptual data model is complete : the database implementation uses tuples whose elements are labels to encode all of its facts.

It is also significant during data modelling : it clarifies the distinction between a name, such as "G 123 XYZ", and the thing being referred to, such as a particular car.

A given set of entities, such as all persons or all cars, is called an **Entity Type**. A given set of labels, such as all registration marks or all surnames, is called a **Label Type**. (A set can be both). The cartesian product consisting of all tuples with a given index set and a given domain function is called a **Fact Type**. Thus, the fact *f* defined earlier belongs to the Fact Type $\Pi(D_r \mid r : R)$. Occasionally the term **Object Type** is used to mean something that is either an Entity Type, a Label Type, or a Fact Type.

Among other things, a conceptual data model declares a set of Entity Types, Label Types, and Fact Types. The domains of each Fact Type are required to be members of the declared set. Any database conforming to the data model must be capable of storing any fact, usually encoded in some way, belonging to any of the declared Fact Types, and no other facts. (An information system may implement several different databases, of course). The Entity Types and Label Types are required to be fixed sets, declared in advance (all possible people, all possible cars, etc.). This might be a convenient fiction, but it is sufficient to describe the proper operation of databases.

Remember that an element of a tuple can itself be one of the tuples specified by the data model. In that case it is a NIAM rule that the element's domain must be one of the Fact Types declared in the data model. When a Fact Type is used as a domain it is said to be **nested** , or **objectified**.

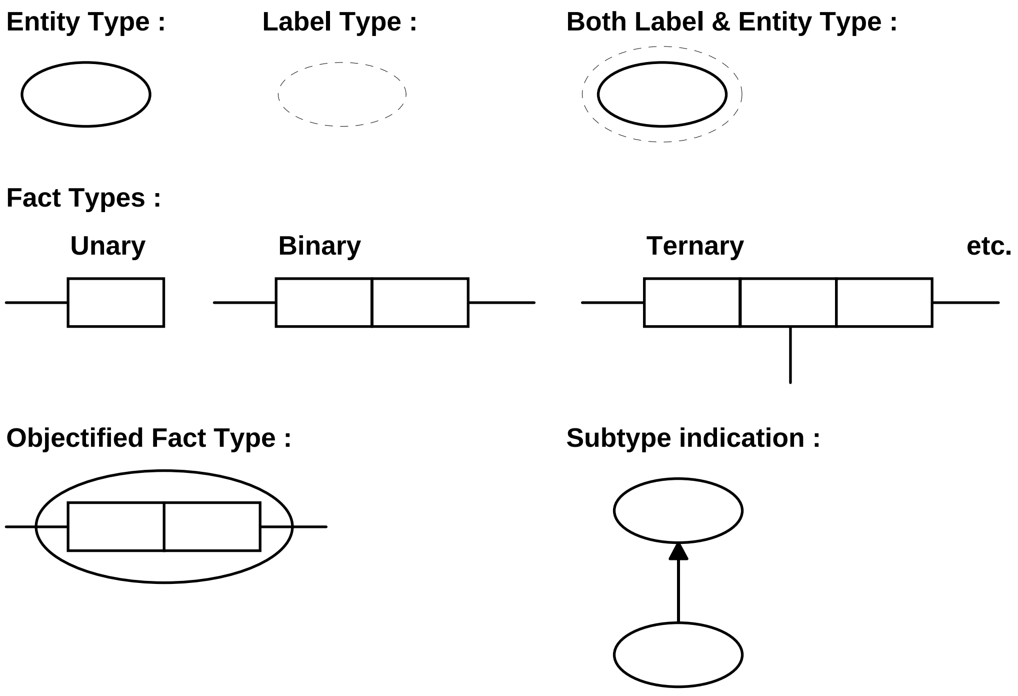
It is a NIAM rule that the Entity Types and Label Types declared in a data model are pairwise disjoint, either naturally disjoint or artificially made disjoint. E.g No person is a car; no height is a weight. It is also a rule that the Fact Types declared in a data model have pairwise disjoint index sets. I.e No role appears in more than one Fact Type, so no role is associated with more than one domain.

Entity Types, Label Types, Fact Types, and roles may be given human-readable names to facilitate communication between people. Typically, most Fact Types and some roles are unnamed.

A conceptual data model is typically displayed as an **information diagram**, alias **ER diagram**, with some supplementary text. The principal symbols in the graphical notation are illustrated in Figures 2.1.1.1 and 2.1.1.2 below. Each Entity Type and Label Type is represented by an ellipse, dotted for Label Types. Each Fact Type is represented by a compound symbol : one or more adjacent boxes with a line from each box to an ellipse. If the Fact Type is nested then there is an ellipse surrounding its boxes. Each box represents one of the Fact Type's roles. The lines say which roles are associated with which domains. Thus the boxes and lines in the compound symbol define the Fact Type's domain function.

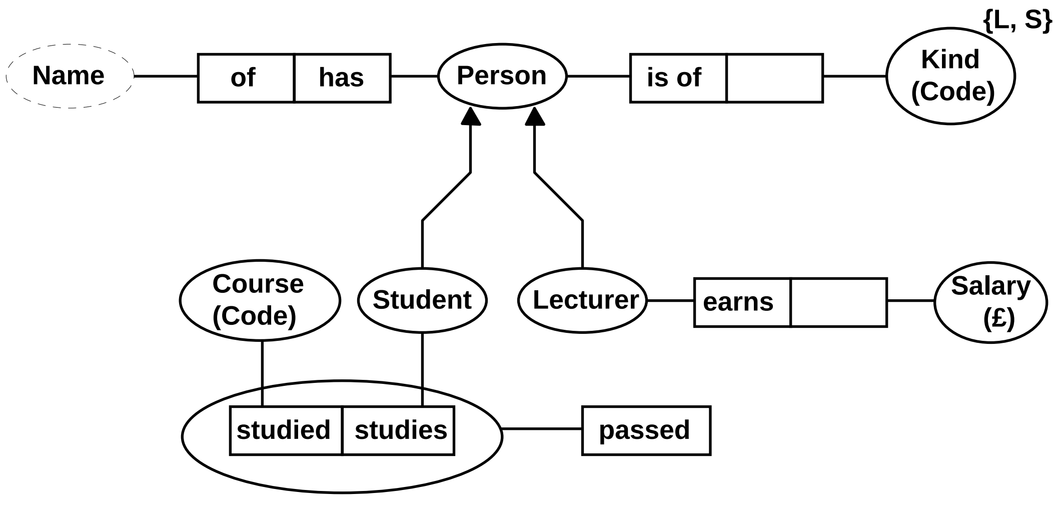
Any human-readable names are written inside the symbols where possible. Figure 2.1.1.1 includes a Subtype symbol; these will be discussed later on.

Figure 2.1.1.1 Principal symbols in the graphical notation



Some common configurations of symbols can be abbreviated. For instance, a Label Type used as the set of identifiers for an Entity Type can be abbreviated to a name in brackets, as in Course (Code) in Figure 2.1.1.2. The membership of a small Label Type can be displayed explicitly in the diagram, as in {L, S} in Figure 2.1.1.2.

Figure 2.1.1.2 An example of the graphical notation



Student = Person of Kind 'S'
Lecturer = Person of Kind 'L'

The database specified by a data model is modelled as an evolving set of facts, alias tuples. Each fact is a member of exactly one of the Fact Types, alias cartesian products, declared by the data model, so the database can also be treated as a set of evolving relations, each with a fixed family of domains and an evolving graph. Thus for each

Fact Type symbol in a data model there is an evolving relation. Given an **instance** of the database, containing a given fixed set of facts, then for each Fact Type symbol there is also a fixed relation.

It is a NIAM rule that any change to the contents of the database can be described as a composition of the two **atomic operations** insert one fact, remove one fact. Of course, the database implementation may make use of different elementary operations provided that the result is always consistent with this rule. Users will often request a change requiring the insertion and removal of several facts where the change is to be treated as a compound **transaction** that is to be done entirely or not done at all. Note that entities and labels are not inserted into or removed from the database except by virtue of being elements of tuples inserted into or removed from the database. ("Remove Carol from the database" is a loose way of saying "Remove all facts about Carol from the database").

There are usually business **rules**, alias **constraints**, concerning facts in the database that the information system can help to enforce. In general, there are two kinds of constraint : static and dynamic. A **static constraint** is a rule that determines whether or not any one database instance is legitimate. For instance, an employee having two salaries is not legitimate. Note that it is quite normal for a change from one legitimate instance to another to pass through instances that are not legitimate. A **dynamic constraint** is a rule that determines whether or not a transition from one database instance to another is legitimate. For instance, reducing the money in a bank account without authorisation is not legitimate. Among other things, a data model declares a set of constraints. The graphical notation includes symbols for the more common forms of static constraint; other constraints are declared in text supplementing the ER diagram.

Most constraints are defined by means of a family, ***pop***, of **population** functions. Each function returns the set of those objects of a certain kind that are mentioned in the given database instance. The functions are as follows. Assume a given data model, and a given instance, *I*, of a database specified by it. Remember that *I* is a set of facts, alias tuples.

$pop_{\mathbf{D}}$: The population of facts in the database instance.
$pop_{\mathbf{D},I} =_{\mathrm{d}} I$.
(Included for convenience).
Note that $pop_{\mathbf{D},I} \subseteq_{\mathrm{d}} \bigcup\{\ G \mid G$ is a Fact Type declared in the data model $\}$

$pop_{\mathbf{F}}$: The population of facts of a given Fact Type, *G*.
$pop_{\mathbf{F},I}(G) =_{\mathrm{d}} \{\ f : I \mid f \in G\ \}$

$pop_{\mathbf{R}}$: The population of objects playing a given role, *r*.
$pop_{\mathbf{R},I}(r) =_{\mathrm{d}} \{\ x \mid \exists\, f : I \bullet r \in \mathrm{Dom}(f) \wedge f_r = x\ \}$

$pop_{\mathbf{E}}$: The population of entities/labels of a given Entity/Label Type, *X*, playing some role.
$pop_{\mathbf{E},I}(X) =_{\mathrm{d}} \{\ e : X \mid \exists\, f : I \bullet \exists\, r : \mathrm{Dom}(f) \bullet e = f_r \wedge$ the domain associated with *r* is declared to be an Entity/Label Type $\}$

Frequent usage is to omit the suffixes, relying on context to imply them. The disjointness rule for index sets ensures that each fact in a database instance belongs to exactly one of the Fact Types declared in the data model, so $pop_{\mathbf{F}}$ returns the desired result. However, there is no *a priori* reason for believing that an entity or label cannot be a tuple, nor that an Entity/Label Type cannot equal a Fact Type. (Consider Complex numbers and Real×Real). Whatever the representation of a data model, for each role it must be possible to determine whether or not it has been declared to be associated with a Fact Type. Given this and that Entity/Label Types are pairwise disjoint, then $pop_{\mathbf{E}}$ is well defined and returns the desired result.

The constraints in any NIAM conceptual data model include two implied constraints. First, in a legitimate database instance, J, any fact $f : J$ must belong to one of the Fact Types declared in the data model, as stated earlier. Second, if a fact $f1$ plays a role in a fact $f2$ then $f2 \in J$ implies $f1 \in J$. Thus, Figure 2.1.1.2 declares that the information system should not allow users to allocate a pass grade to Carol in Physics unless 'Carol studies Physics' is a fact held in the database.

A frequent occurrence in data modelling is the need to express a constraint of the form 'A member x of the Entity/Label Type X is permitted to play the role r only if condition α is true', where α is a formula of the form '$x \in pop_{\mathbf{R},\,I}\,(r1) \ \wedge \ldots$'. For instance, in Figure 2.1.1.2, a fact that the person Carol studies some course is allowed in the database only if the fact that Carol is of Kind "S" is also held in the database. Such an evolving subset, determined by α, of an evolving subset, $pop_{\mathbf{E},\,I}\,(X)$, is called a **Subtype** (an unfortunate name). The graphical notation uses an ellipse to represent a Subtype, and an arrow to point to the symbol representing the domain, which, in practice, is usually an Entity Type. Both Student and Lecturer are Subtypes in Figure 2.1.1.2. Alternatively, the arrow may point to another Subtype symbol to show that α is a restriction of another formula, or there may be several arrows pointing to other Subtype symbols to show that α is a restriction of the conjunction of other formulas. An additional population function can now be defined.

$pop_{\mathbf{S}}$: The population of entities/labels of a given Subtype with domain X and formula α.
$pop_{\mathbf{S},\,I}\,(X, \alpha) =_{\mathrm{d}} \{\, e : pop_{\mathbf{E},\,I}\,(X) \mid \alpha \,\}$

The ellipse representing a Subtype is best thought of as being a copy of the symbol representing its superior Entity/Label Type. Any name in the ellipse identifies the defining formula, which is written elsewhere. A line from a role box to a Subtype symbol identifies both a domain and a constraint. The arrow from the Subtype symbol points towards the master declaration of the domain associated with the role. The constraint for the role r and the Subtype with domain X and formula α is the rule declaring that for any legitimate database instance $pop_{\mathbf{R},\,I}\,(r) \subseteq pop_{\mathbf{S},\,I}\,(X, \alpha)$.

A Subtype is sometimes said to **inherit** all the Fact Types that make use of its superiors, but note that this is a statement about types, not about members. A student such as Carol has a height and a weight by virtue of also being a person, not by inheriting them from another entity.

It is a NIAM rule for ER diagrams that all transitive links in a subtype structure are omitted; that is, if x is a Subtype of y and y is a Subtype of z then there is no arrow going from the symbol for x to the symbol for z. This presumably makes the diagrams easier to read. It is also a rule that no Subtype defining formula constrains a Subtype population to be empty for all database instances, nor equal to the population of another Subtype or Entity/Label Type for all database instances.

A **total role constraint** is a rule declaring that for any legitimate database instance every member of a certain population plays a certain role. For instance, a rule that all employees mentioned in the database must have their salaries recorded. There are three cases in the graphical notation. When the sub-symbol for the role r is connected to the symbol for the Fact Type G then the rule is $pop_{\mathbf{R},I}(r) = pop_{\mathbf{F},I}(G)$; when to the symbol for the Entity/Label Type X then $pop_{\mathbf{R},I}(r) = pop_{\mathbf{E},I}(X)$; when to the symbol for the Subtype with domain X and formula α then $pop_{\mathbf{R},I}(r) = pop_{\mathbf{S},I}(X, \alpha)$. Note that the word "total" refers to a population, not to a domain. A Type or Subtype is said to play a **mandatory** role when subject to a total role constraint; otherwise it is said to play an **optional** role. The graphical notation includes a way of declaring this kind of constraint.

An **intra-fact uniqueness constraint**, sometimes abbreviated to **uniqueness constraint**, is a rule declaring that for any legitimate database instance the population of a certain Fact Type must form a relation that is functional in nature. Specifically, the rule states that for a certain Fact Type, G, and a certain subset, K, of its index set, R, then for any legitimate database instance, I,

$$\forall f1, f2 : pop_{\mathbf{F},I}(G) \bullet K \langle f1 = K \langle f2 \Rightarrow f1 = f2 .$$

The subset K acts like the key in a database table. The case $K = R$ is still regarded as a uniqueness constraint even though it does not flag any populations as illegitimate; it is the default case if no tighter constraints are known. The graphical notation includes a way of declaring this kind of constraint.

A non-unary Fact Type can have more than one uniqueness constraint. For instance, if its index set is $\{r1, r2\}$ and the legitimate populations form an evolving injection then there are two uniqueness constraints : one specified by $\{r1\}$, the other by $\{r2\}$.

It is a NIAM rule that every fact specified by a conceptual data model is an **elementary fact**. That is, no fact holds two or more independent items of information. For instance, 'Carol studies Physics and is 1.70m tall' is not an elementary fact if her height has nothing to do with the subject she studies. Of course, it would be an elementary fact if we wish to remember which subjects she studied as she grew taller.

One consequence of this rule is that the evolving relation associated with each Fact Type is in 5th Normal Form (no join dependencies). More accurately, for each Fact Type, G, there is at least one legitimate database instance, I, for which $pop_{\mathbf{F},I}(G)$ is the graph of a relation in 5th Normal Form. Another consequence of the rule is that the uniqueness constraints can be used in a plausibility test. If a Fact Type with n roles has a uniqueness constraint spanning fewer than n - 1 roles then its facts are not elementary.

A **derived Fact Type** is a Fact Type subject to a rule that for any legitimate database instance the Fact Type's population is uniquely determined by the populations of other Fact Types. For instance, a rule that NetPrice(x) = GrossPrice(x) + Markup(x) for any Item x whose gross price and mark-up are both recorded in the database. Conceptually, facts belonging to a derived Fact Type are stored in the database, but in practice they will often be calculated on demand without being stored. Although the derivation rule constrains a single Fact Type, it can often be used as a way of describing a constraint on a set of Fact Types. For instance, each of NetPrice, GrossPrice, and Markup is constrained by the other two.

There are no restrictions on the rules that can be used to define derived Fact Types provided their populations are always well defined. This implies that their populations must ultimately be derived from the populations of non-derived Fact Types.

The graphical notation includes symbols for several more kinds of constraint; they are not described here. In general, every known constraint should be declared in the conceptual data model, regardless of whether there is a defined symbol for it in the graphical notation. The list of possible classes of constraint is endless.

There is often more than one way of modelling a given part of the real world. For instance, an appointment can be described as a primitive entity, or as a tuple combining person, time, and place; the unary fact 'George smokes' can be represented by the binary fact 'George smokes?, answer Yes'; there can be a choice as to which Fact Type is flagged as being derived. Converting from one representation to another is called **schema transformation**. The [NH] book describes several common cases, but their justification is informal and for the more complicated cases the book gives no general rules for checking correctness.

There is an algorithm which will convert any well formed NIAM conceptual data model into a relational database design that has the property of being in **Optimal Normal Form**, alias **ONF**. Each table in the resulting database is [LN], or is typically [NH], in fifth normal form, and the number of tables is minimal. Note that normal forms are defined with respect to the declared constraints; the quality of the database design depends very much on how well business rules have been translated into constraints by the data modellers. Note also that designers using NIAM have no need to apply normal form theory in order to produce satisfactory database designs.

In outline, the algorithm first transforms each non-derived Fact Type into a database table, then merges tables where this would be reasonable. For instance, if each student studies exactly one subject and has exactly one height then the independent facts 'Carol studies Physics' and 'Carol is 1.70m tall' can be held as (Carol, Physics, 1.70) in one row of a combined table.

The [LN] paper includes the description of an application program, **AREDD**, that enables a user to enter the details of a data model one element at a time, and, subject to passing quality checks, output the corresponding ONF database design. Apparently, (the text is not entirely clear), as each Fact Type is declared the program checks that its domains have already been declared. This ensures that the data model has a sensible Fact Type structure, and implies that the class of permitted data models is to this extent inductively generated. Curiously, this incremental checking technique is not used for Subtype structures; they are not checked until the user requests a global check of the data model.

Both [LN] and [NH] include more or less complete data models, alias **meta-schemas**, of data models. One of them specifies the database used by the AREDD program described above. Nijssen and his colleagues believe that it is important for a data modelling language to be able to describe its own structure.

Comments

The [NH] book is to be commended for its general clarity, and also for its dedication to the principles that practitioners should understand what they are doing and should be given techniques that will encourage them to produce good designs.

The definitions of "entity" and "Entity Type" are confusing, differ between [LN] and [NH], and lead to the use of awkward circumlocutions. It would be better to say that every primitive object is an entity, that a label is a certain kind of entity, and that the generic term for Entity Type or Fact Type is Object Type.

Representing tuples as distributed functions with choosable index sets has several advantages. No special definition is needed for unary (or nullary) cartesian products. The representation models the belief that "the dog chased the ball" and "the ball was chased by the dog" have essentially the same information content. Cartesian products with different index sets are automatically disjoint; no additional indexes are needed to distinguish one Fact Type from another.

Merging an Entity Type symbol with its identifying Label Type symbol, as in Kind (Code), is really an implementation step, but one which makes the diagram easier to read.

The [NH] book sometimes uses a Subtype arrow to indicate that one Label Type is a fixed subset of another Label Type. This is a misuse of the arrow symbol since it is clear that in these cases the populations of the two Label Types vary independently, rather than one population being constrained by the other.

Figure 8 in [LN] contains the ER diagram for the database used by the program AREDD. It declares that every legitimate data model has at least one Entity/Label Type, Fact Type, role, and constraint. This is a sensible requirement for a data model that is about to be converted into a database design. However, it is not a suitable requirement when information about a data model is being entered incrementally. Why shouldn't the user be allowed to log out after entering just an Entity Type?

This illustrates a general problem with databases that hold design information. It is likely that the completed design must obey extra constraints that do not apply to earlier stages of the design. For instance, a Pascal program must end with a full stop, but an incomplete program need not. NIAM does not distinguish between constraints that apply always and those that do not. That is, NIAM, and other database design techniques it seems, do not distinguish between legitimate draft database instances and legitimate final database instances.

2.1.1.1 - Second edition Halpin [1995]

References

Halpin, T A [1995]. Conceptual schema & relational database design (2nd Ed).
Prentice Hall of Australia Pty Ltd.
Library reference : Joule 001.6442/HAL.

Summary

Halpin [1995] is the second edition of Nijssen and Halpin [1989]. It has been extensively revised, and contains far more comparisons with Chen-style ER data modelling (described below in Section 2.2.2) and less discussion of general system design.

More acronyms are introduced :

ORM	**Object Role Modelling**. A general class of conceptual data modelling techniques that includes NIAM and FORM (see below).
NIAM	Now said to stand for **Natural language Information Analysis Method**. NIAM is declared to be the design method described in the first edition.
FORM	**Formal Object Role Modelling**. FORM is declared to be the method (notation and techniques) described in this second edition.
FORML	Pseudo-English text-based language that is part of FORM.

There are some changes in terminology :

Labels and Label Types are now called **values** and **Value Types**.

Every Fact Type now has an associated Predicate. A **Predicate** is defined to be a "sentence with object-holes in it" (p47 & 538).

The algorithm for converting a data model into a relational database schema is now named **Rmap**. It is discussed in much more detail than in the first edition.

There some other changes :

Each Fact Type is named by its associated Predicate. There is required to be an order-preserving bijection between the holes in a Fact Type's Predicate and its roles.

Roles no longer have human-readable names.

It is no longer possible to have an Entity Type that is also a Label Type.

There are some additional diagram symbols.

Comments

The terms Fact Type and Predicate are used in ways that may confuse some readers into thinking that they are synonyms. For instance, the phrase "objectified predicate" is used.

Moreover, the section that introduces Fact Types does not mention that a Fact Type is a set of facts. The reader must remember that an earlier section defined "Type" to mean "a set of all possible instances".

The definitions of entity and label, alias value, have changed yet again and are still confusing. A value is defined to be an "unchangeable object that is identified by a constant", and an entity is defined to be an "object that is described (not a value)" (p537-538). There is no useful difference between "identified" and "described", and if a database about people is to be of any use to its owners then a person must be just as constant as 1.414. In addition, facts belonging to objectified Fact Types are now also classified as entities.

2.1.2 The Predicator Model - van Bommel et al [1991]; van der Weide et al [1992]

References

[BHW] van Bommel, P, ter Hofstede, A H M, and van der Weide, Th P [1991]. Semantics and verification of object-role models.
Information Systems, Vol 16, No 5, p471-495.

[WHB] van der Weide, Th P, ter Hofstede, A H M, and van Bommel, P [1992]. Uniquest : Determining the semantics of complex uniqueness constraints.
The Computer Journal, Vol 35, No 2, p148-156.

Highlighting

Special terms defined in the papers are highlighted in bold at the point where they are introduced.

Summary

The [BHW] paper is the first of many papers from the University of Nijmegen that define and use the **Predicator Model**, a model of the kind of conceptual data models used in NIAM. The paper claims to be the first formal description of NIAM and related styles of data model, a claim which appears to be entirely justified. The [WHB] paper is effectively an appendix, providing a more detailed description of an algorithm. To do this, it includes a succinct one-page definition of the predicator model ([WHB], p149).

The [BHW] paper has four objectives.

1) To construct a model of conceptual data models and the databases they specify.
2) To define the meaning of the usual NIAM constraint symbols.
3) To investigate the problem of entity identification.
4) To define some quality checks on data models, and determine the complexity of algorithms that apply these checks.

This summary describes each objective under a separate heading, preceded by some preliminaries.

Preliminaries

These and related papers use symbols that cannot be properly reproduced here. Substitute symbols are used instead; see Figure 2.1.2.1.

Figure 2.1.2.1 Substitute symbols

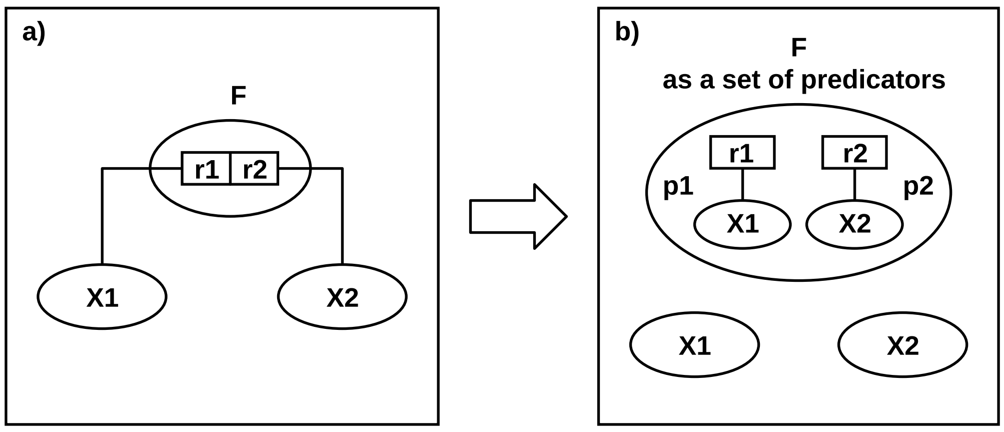
Symbol	Substitute
$\mathcal{A}, \mathcal{C}, \mathcal{E}, \mathcal{F}$	***A, C, E, F***
$\mathcal{I}, \mathcal{L}, \mathcal{O}, \mathcal{P}, \mathcal{R}$	***I, L, O, P, R***
$\sqcap$	***Top***
$\models$	***Satisfies*** **(their usage)**

The introduction, sec 2.1 of the [BHW] paper, uses two phrases which might be unfamiliar. A **tuple oriented approach** refers either to the representation of tuples as nested couples, as when the tuple (a, b, c) is represented by $\langle\langle a, b\rangle, c\rangle$, or perhaps to their representation as families with predetermined index sets, as when (a, b, c) is represented by the family (t_1, t_2, t_3). A **mapping oriented approach** refers to the representation of tuples as families using any convenient index set, as when (a, b, c) is represented by the family $(t_i \mid i : I)$. The papers use the latter representation.

The Predicator Model is named for its use of objects called **predicators**. The introduction, sec 2.1 of the [BHW] paper, describes the internal structure of these objects, but as it happens this internal structure is not used in the model itself. Predicators are treated as primitive objects. This is just as well, as there would be severe technical problems if the description of the internal structure were to be taken literally (see the comments following this summary). The description of predicators given here should be ignored when reading the description of the Predicator Model.

Consider the tiny data model shown in Figure 2.1.2.2(a). It is to be treated as a collection of symbols which are connected together in a particular way. Any meaning it might have is to be given separately. The line from the role symbol *r1* to the symbol *X1* indicates an association between *r1* and *X1*. This association is called a predicator. Similarly, there is a predicator associating *r2* with *X2*. It is possible to rearrange the picture without losing any information to give Figure 2.1.2.2(b). The Fact Type symbol *F* can now be modelled as a set, {*p1*, *p2*}, of predicators. Note that *X1* and *X2* could also be Fact Type symbols with their own internal structures.

Figure 2.1.2.2 Predicators



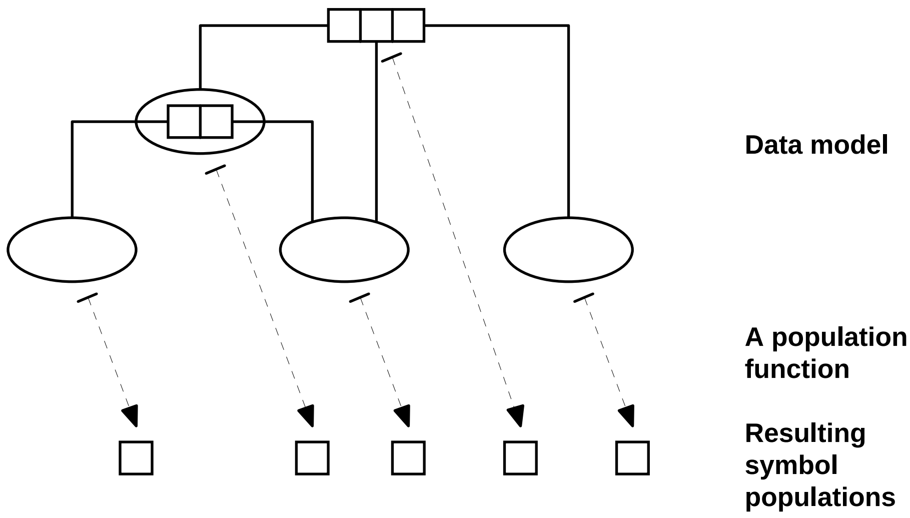
Description : The model of data models and databases

The first objective of the [BHW] paper is to construct a model, the Predicator Model, of conceptual data models and the databases they specify. The model is defined in three stages. The first stage describes the structure of data models; the second stage describes a model of databases; and the third stage adds database constraints.

The general modelling technique used by them will be described first. They use the common technique of describing a single generic data model, and representing it as a tuple whose elements are various sets and relations. Each Entity Type, Label Type, and Subtype symbol in the data model is modelled as a primitive set, with no defined internal structure. Note that these sets are abstract symbols, not domains. Each Fact Type symbol in the data model is modelled as a set of predicators; each of these sets is also an abstract symbol. Each predicator is modelled as a primitive set, whatever the introduction might have said. Lines connecting symbols together in the data model are modelled by the graphs of relations.

The database instances relevant to the data model are modelled by a class of population functions. Each function assigns some set to each symbol in the data model, as illustrated in Figure 2.1.2.3. The sets assigned by a single function collectively represent a single database instance. A combination of implied constraints and constraints given expressly in the data model distinguish good population functions from bad ones.

Figure 2.1.2.3 A data model and its database populations



Now for the details. They are introduced here in an order that is different from that in the papers. Assume that a particular data model is given, such as the example shown below in Figure 2.1.2.4(a).

Each Entity Type symbol, Label Type symbol and Subtype symbol appearing in the data model is modelled by a primitive set. They are collected together into the sets ***E*** and ***L***. *E* is the set of **entity types** in the data model; *L* is the set of **label types** in the data model. It is required that *E* and *L* are disjoint, so combined Entity & Label Type symbols are not modelled. Their union is the set $\boldsymbol{A} =_{\mathrm{d}} E + L$ of **atomic object types**. Each Subtype symbol is a member of *A*, and so is classified either as an entity type or as a label type. Note that in the [BHW] paper the terms entity type, label type, and atomic object type refer to symbols, not to designated domains such as Person or Nat. Note also that *A* can be infinite, but must be finite if the data model is to have certain useful properties.

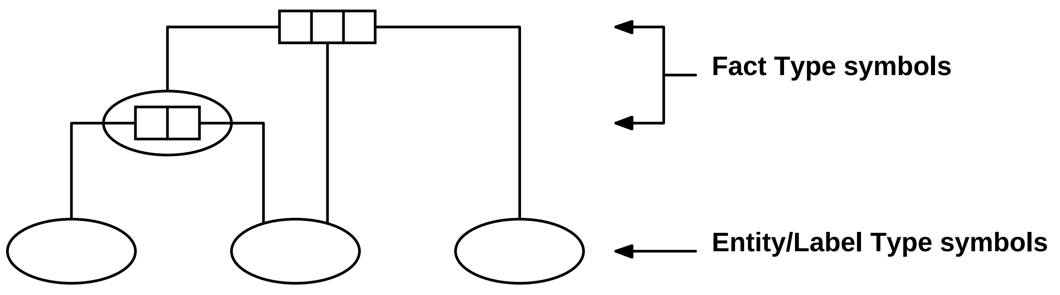
Each Fact Type symbol in the data model is modelled by a non-empty finite set of predicators. (Later on, these sets of predicators will be used as index sets of tuples). The function **Base** associates each predicator with a symbol in the data model, thus modelling the lines in the ER diagram. Figure 2.1.2.4 below shows this translation of a data model into its Predicator model.

The sets modelling Fact Type symbols appearing in the data model are collected together into the set ***F*** of **fact types**. Thus *F* is a set of sets of Predicators. It is required that *F* is finite and pairwise disjoint and that *F* and *A* are disjoint. The set $\boldsymbol{P} =_{\mathrm{d}} \bigcup F$ is the set of predicators occurring in the data model. Note again that the term fact type refers to symbols, not to designated sets of tuples.

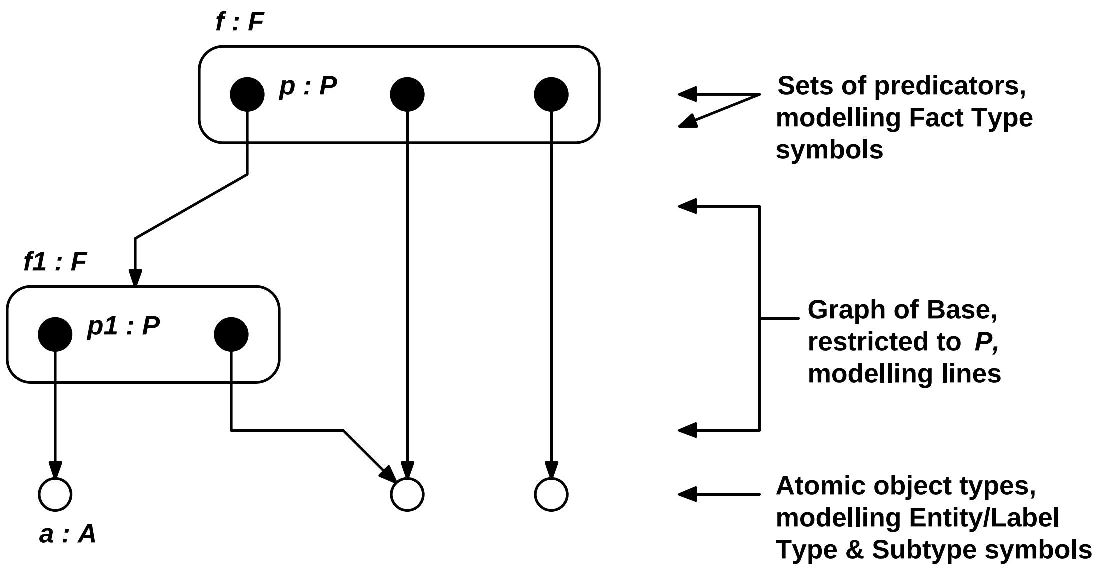
The set of all type-symbols in the data model is the set $\boldsymbol{O} =_{\mathrm{d}} E + L + F$ of **object types** occurring in the model. For each predicator $p : P$ there is the requirement that Base(p) $\in O$. Since *F* is pairwise disjoint, *p* is a member of exactly one fact type, $f : F$; the function ***Fact*** returns *f* given *p*. The Predicator Model makes use of predicators that are not members of *P*. For these predicators Base is defined but not *Fact*.

Figure 2.1.2.4 Picture of a simple data model and its Predicator model

a) Data model
(names are not modelled)



b) Its Predicator model : *A, F, Base, Fact*


f : F
p : P
Sets of predicators, modelling Fact Type symbols
f1 : F
p1 : P
Graph of Base, restricted to P, modelling lines
a : A
Atomic object types, modelling Entity/Label Type & Subtype symbols

A	**- set of atomic object types**
F	**- set of fact types**
P	**- set of predicators**

Fact (p)	***= f***
Base (p)	***= f1***
Base (p1)	***= a***

The data model can contain Subtype symbols, as in the example shown below in Figure 2.1.2.5 (a).

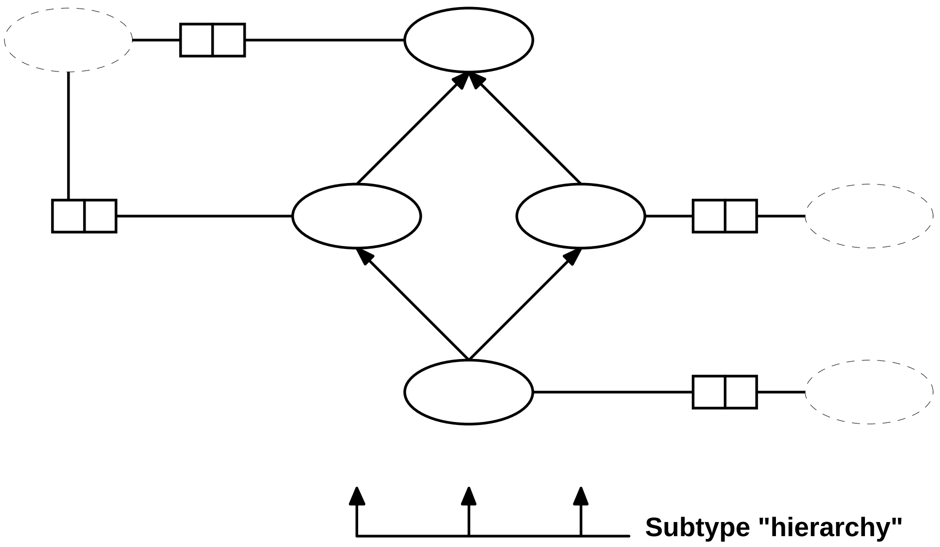
The occurrence of Subtype symbols in the data model is modelled by the relation *Sub* : $O \leftrightarrow O$, where *a Sub b* means that *a* is a subtype of *b*. There are several constraints on *Sub*. It must be a strict partial order relation; i.e *Sub* is anti-reflexive and transitive. Its graph is constrained by Gr(*Sub*) $\subseteq_d E \times E \cup L \times L$. Finally, each object type $x : O$ must either be a maximal element of *Sub*, or related to exactly one maximal element of *Sub*. Note that Figure 6 in [BHW], and its associated text, illustrate *Sub* by describing a relation that is anti-transitive. *Sub* is its transitive closure.

Given any object type *x* : *O*, the function ***Top*** returns *x* itself if *x* is a maximal element of *Sub*, otherwise it returns the unique maximal element *m* such that *x Sub m*. *Top*(*x*) is called the **pater familias** of *x*. Figure 2.1.2.5 below shows this translation of a data model into its Predicator model.

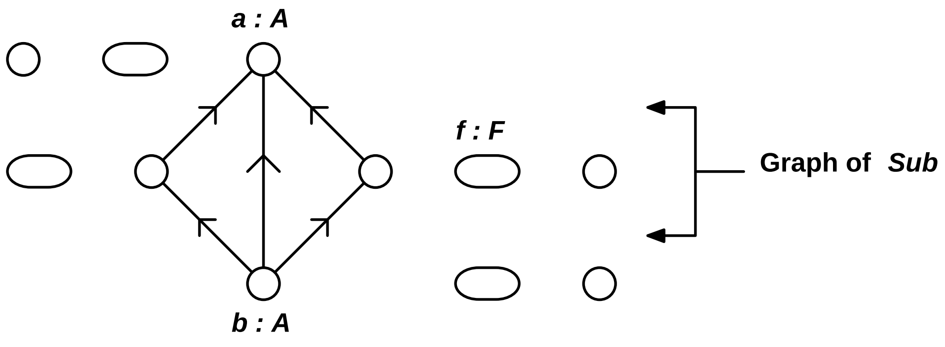
The equivalence relation ~ is defined for all predicators *p*, *q* with Base(*p*), Base(*q*) : *O* by $p \sim q \ \Leftrightarrow_d \ Top(\text{Base}(p)) = Top(\text{Base}(q))$. If *p* ~ *q* then *p* and *q* are said to be **attached** to each other. Note that fact types cannot have subtypes and that each is its own pater familias.

Figure 2.1.2.5 Picture of another simple data model and its Predicator model

a) Data model with subtype symbols



b) Its Predicator model : *A, F, Sub, Top*



A **- set of atomic object types**
F **- set of fact types**

Top (a) = a
Top(f) = f
Top (b) = a

All these components are now collected together into an **information structure** represented by the tuple $\boldsymbol{I} =_d (P, O, Sub, F)$ [BHW] or $\boldsymbol{I} =_d (P, O, Sub, F, \mathrm{Base}, Top)$ [WHB].

This completes the first stage of the Predicator Model of a data model. To sum up, certain components of the ER diagram are modelled by the component parts of *I* :- Fact Type symbols by sets of predicators; Entity Type, Label Type, and Subtype symbols by primitive sets; lines by the graph of the function $P \langle \mathrm{Base}$; and subtype arrows, or, more accurately, their transitive closure, by the graph of the relation *Sub*.

The description now turns to the model of databases. Given an information structure *I*, a database **population** is any function, ***Pop***, that assigns a set to each member of *O*; i.e $Pop : O \rightarrow \mathrm{Set}$. (Note : later versions of the Predicator Model assign members of a certain set). Obviously, such an unrestricted assignment can include silly cases. The predicate **IsPop** is used to pick out the useful populations that obey the fact type and subtype structure of *I*, namely the populations where

a) the population assigned to each Fact Type symbol is a set of tuples, each having an appropriate index set and elements;

b) the population assigned to each Subtype symbol is a subset of the population(s) assigned to its supertype symbol(s).

Specifically, IsPop(*I*, *Pop*) is true iff

a) for each $f : F$ every $t : Pop(f)$ is a tuple $(t_p : Pop(\mathrm{Base}(p)) \mid p : f)$; (remember that *f* is a set of predicators)

b) for each $a, b : A$, $a\ Sub\ b$ implies $Pop(a) \subseteq Pop(b)$.

Thus when IsPop(*I*, *Pop*) is true the set of predicators modelling a Fact Type symbol is the index set for the tuples allocated to that symbol. Note that most of the [BHW] paper and all of the [WHB] paper assumes that IsPop(*I*, *Pop*) is true. Note also that any *Pop* describes a single database instance. Just as the model of data models concentrates on a single generic instance, so does the model of databases.

Two other predicates are occasionally used or inferred. First, ***Connected***(*Pop*) (*I* is apparently implicit) is true iff the population of each atomic pater familias equals the union of its role populations. In other words, iff the population of Entity and Label Types is determined entirely by the facts, alias tuples, held in the database. Second, *Pop* obeys the **strong typing rule** iff the populations of atomic pater familias are pairwise disjoint. (This is a NIAM rule).

The model of data models is now completed by adding a set, ***C***, of **constraints** to the information structure *I* to form the **schema** $\Sigma =_d (I, C)$. Each constraint $c : C$ is a formula (of set theory). A population function *Pop* obeys the constraint *c*, written *Pop* ***Satisfies*** *c*, iff *c* is true given *Pop*. The predicate **IsPop** is extended to schemas in the obvious way :

$$\mathrm{IsPop}(\Sigma, Pop) \Leftrightarrow_d \mathrm{IsPop}(I, Pop) \ \wedge\ \forall\, c : C \bullet\ Pop\ Satisfies\ c$$

The database instance described by *Pop* is legitimate iff IsPop(Σ, *Pop*).

Description : The meaning of constraint symbols

The second objective of the [BHW] paper is to define the meaning of the constraint symbols used in the NIAM notation. The technique they use is illustrated here by the definition of one kind of constraint symbol. A generalised **total role constraint** symbol selects a set $\tau \subseteq_d P$ of predicators. τ acts as a specifier for the constraint. The function ***total*** converts this specifier into a formula, which is now declared to be a member of C, the set of constraints. *total* is defined only for subsets of P whose members' bases have a common pater familias. The definition in the paper uses a special language that will be outlined in a moment. In a more conventional language the definition is :

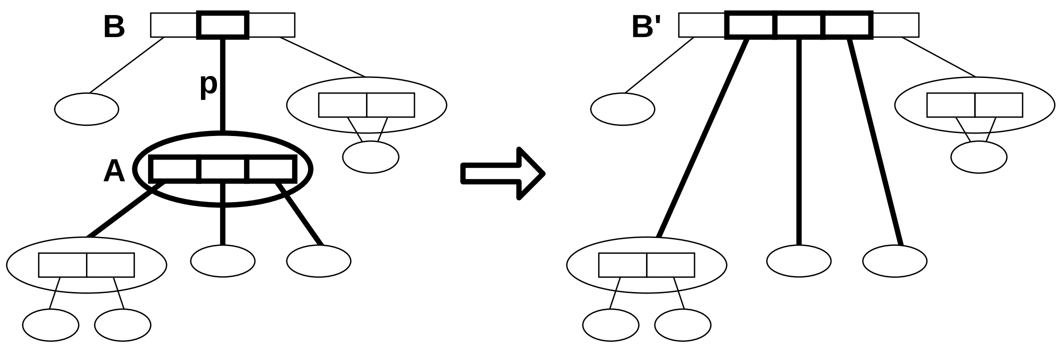
$$total(\tau) =_d \text{'} \bigcup_{q:\tau} Pop(\mathsf{Base}(q)) = \bigcup_{q:\tau} \{ x \mid \exists t : Pop(Fact(q)) \bullet x = t_q \} \text{'}$$

A specifier and function, similar to τ and *total*, and an appropriate precondition, are defined for each kind of constraint symbol used in the NIAM notation. In addition, Subtype defining rules are replaced by constraints. (Remember that a Subtype symbol's population is given by *pop*; it is not a defined subset of some other population). There is a function, ***SubRule***, which associates each Subtype symbol belonging to A with the constraint specifier that takes the place of the Subtype's defining rule. The corresponding constraint formulas are also declared to be members of C.

Constraint formulas make use of a freely generated class, $\mathsf{R}(I)$, of **relational expressions**, such as the expression $\sigma_\alpha \pi_{a:p,\, b:q} f$ which describes a selection operation acting on the results of a projection operation. These expressions are also, somewhat confusingly, called **derived fact types**, meaning that they are derived from the members of F. Two functions, ***Pop*** and ***Schema***, on $\mathsf{R}(I)$ are defined recursively. Given any relational expression and a population function *Pop* obeying $\mathsf{IsPop}(I, Pop)$, the function *Pop* (extended) returns a set of tuples, the value of the expression; the function *Schema* returns the set of predicators that is the index set of these tuples. Several non-recursive functions are also defined on $\mathsf{R}(I)$. Some of the definitions implicitly assume that each member of O has been assigned a finite population.

The alphabet of $\mathsf{R}(I)$ uses several operator symbols taken from the theory of relational databases : join, projection, selection, etc. *Pop* is defined accordingly. Some of the operators can introduce predicators that do not belong to P, though their bases must still belong to O. One of the operators, μ (unnest), is rather unusual. It acts on a single objectified fact type in the manner illustrated in Figure 2.1.2.6 to do one step of unnesting. Note that, despite appearances, μ and the other operators transform sets of tuples, not data models.

Figure 2.1.2.6 The action of the unnest operator



Description : The problem of entity identification

The third objective of the [BHW] paper is to determine whether a given data model can be implemented in a practical database system. It can be if every legitimate database instance can be encoded unambiguously as a set of tuples that use nothing but labels. (Note : the [BHW] paper does not phrase it this way; a more roundabout description is used.)

The question is one of **identification** : can labels be used to identify entities and tuples? Most identification schemes are very simple. For instance, employees are often identified by employee numbers. For this to work correctly, every current employee must have a number, and the association of current employees with numbers must be an (evolving) injection. More complicated identification schemes can arise. The [BHW] paper, in Figure 26, p487, gives an example where each house is identified by a house number and a street, which is identified by a street name and a community, which is identified by a community name. In every case, identification requires an appropriate combination of data model structure (e.g current employees have numbers), total role constraints (e.g every current employee has a number), and uniqueness constraints (e.g each current employee has at most one number).

The [BHW] paper defines a rule for deciding whether a data model is implementable. When it is, the data model is called **structural identifiable** [sic]. Suppose that a data model, described by the schema $\Sigma =_d (I, C)$, is structural identifiable, then it and its legitimate populations have certain properties. One property is that the set *A* of atomic object types contains a finite number of label types and a finite number of entity types that are paterfamilias. Each of these label and entity types must also be the base of some predicator belonging to *P*. Curiously, the number of entity types that are Subtype symbols can be infinite. (Corollary 5.1 in [BHW] p486 suggests otherwise. This may be an oversight or a printing error.) Another property is that the population of each atomic object type is entirely determined by the fact type populations.

Yet another property concerns the structure of the information structure *I*. Suppose we choose a predicator *p* belonging to the fact type *f* : *F*. If the base of *p* is the fact type *f* ', then choose a predicator *p*' : *f* '. If the base of *p*' is the fact type *f* '', then choose a predicator *p*'' : *f* '', and so on. Since the set *P* of predicators is finite, this process of choosing must either end with a predicator whose base is not a fact type, or will revisit a fact type already seen. If there is a sequence of choices that would cause it to revisit a fact type then *I* is said to be **cyclic**. The property is that *I* is not cyclic. Moving from a predicator to its base always brings one nearer to an atomic object type. Remark : This is a respectable property, but its justification in [BHW] is specious; this is discussed later on.

Description : The complexity of some quality checks

The fourth and last objective of the [BHW] paper is to investigate the difficulty of doing a quality check on the data model's constraints. It may be that the constraints are so tight that there is a fact type whose only legitimate population is $\varnothing$ (an exclusion constraint can have this effect). If so, either the data modeller has made a mistake, or there is an unnoticed conflict in the business rules. The complexity of any algorithm for making this quality check is obviously of interest.

The problem to be solved is "Given any schema Σ, say yes if there is a population function *Pop* for which IsPop(Σ, *Pop*) and also *Pop*(*f*) ≠ ∅ for every *f* : *F*; otherwise say no". The size of the problem is the number of constraints, *Num*(*C*). The [BHW] paper shows that the problem's complexity is at least NP-Complete. If IsPop(Σ, *Pop*) can be evaluated in polynomial time with respect to *Num*(*C*) then it is exactly NP-Complete. This will be so if

a) Every label type has a constraint that ensures its population is finite and bounded. E.g As in "Every bank balance < 10^{20} ¥ ".

b) The data model is structural identifiable (so implying a finiteness and boundedness constraint on each entity type population).

c) Every constraint formula, including Subtype defining formulas, is decidable (which is so for constraints specified by the usual NIAM symbols).

[BHW] fails to make points (b) and (c). A useful algorithm could concentrate on constraints specified by NIAM symbols, assuming that all other constraints are simple, which they usually are. The algorithm's complexity would still be NP-Complete.

Comments

Comments on the Predicator Model have been numbered as there are so many of them. They have also been classified as criticisms ("Problem"), or as highlighting features that make the model unsuitable for our purpose ("Weakness"), or, finally, as conclusions ("Conclusions").

Incidentally, part of the Predicator Model is translated into Scheurer's Feature Notation in Section 3.3.

Problem 1
Readers of these papers are warned than many operator symbols, including "=", are overloaded or extended, often without warning. The detailed definitions are well intentioned, but, as printed, not always fully defined. It is not always apparent where finiteness restrictions are introduced.

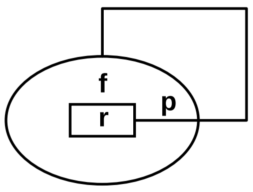
Problem 2
[BHW] p475 states that the database contents and their evolution is isomorphic to a part of the real world. In other words, databases always tell the truth. This is a very bad thing to say to practitioners and students. The information put into any database is subject to human error and sometimes to criminal falsehood. This should be understood from the very beginning of the system design process. Error correction procedures should be regarded as a normal part of the requirements, not as an arbitrary implementation decision. Mistakes in your bank account should be corrected by means of the proper database update process, not by a low level disc editor.

Problem 3
The information structure *I* is defined in [BHW] to be the tuple (*P*, *O*, *Sub*, *F*). This loses the distinction between label types and entity types. A minimal definition of *I* would be (*E*, *L*, *F*, Base, *Sub*). The two papers have different definitions of *I*. This does not inspire confidence in the Predicator Model.

Weakness 1
Predicators are not introduced. They may form a proper class, in which case the class, R(*I*), of relational expressions may be a freely generated proper class with a recursion theorem that speaks of class functions. This is not improper, but it is an unnecessary complication. It would be tidier to introduce a set, Predicators. It need be no more than countably infinite.

Problem 4
There is another problem with predicators. In the introduction to the model, each predicator is described as being a couple, with one element being a role, the other being an object type symbol. This notion must be discarded in the Predicator Model. Each fact type symbol is a set of predicators, and cyclic information structures are permitted, such as the structure shown in Figure 2.1.2.7 below. The introductory description of predicators would require the existence of a predicator $p =_d \langle r, \{p\} \rangle$, but there is no such set. Thus, the Predicator Model cannot use "predicators"! However, each predicator used in the model requires no declared internal structure, is associated with an object type symbol by the function Base, and is used as an index in tuples. It would be possible to avoid some misleading preconceptions by renaming the class Predicators as Roles.

Figure 2.1.2.7 The cyclic structure from [BHW] Figure 28, p487



Alternatively, if the model is to make explicit use of predicators as originally described then

a) The information structure *I* must not be cyclic, as a precondition;

b) It must be recognised that each predicator whose base is a fact type symbol holds an encoded version of the structure of that fact type.

We must conclude from (b) that *I* can be partially redundant, and that we would need some convenient means of describing the structure of predicators, one not using predicators.

Weakness 2
The treatment of subtypes is rather weak. *Sub* is a transitive relation whereas a standard NIAM quality check requires that subtype arrows be anti-transitive. A subtype defining rule is required to be a constraint that is specified by a relational expression. The expression must make an overt reference to a superior of the subtype, but not necessarily to an immediate superior, nor to all of its immediate superiors. Consequently there is little connection between the expression's overt structure and the structure of the subtype arrows. Some reasonable subtype definitions cannot be expressed; for instance, people whose bank balance history is a non-decreasing function of time. Fact Types are not allowed to have Subtypes (but later versions of the Predicator Model do allow it).

Weakness 3
The legitimate populations of Label Type symbols are restricted to finite domains (by means of, possibly optional, constraints), though this finiteness restriction changes in later versions of the Predicator Model.

Entity Type symbols are not associated with fixed domains, in contrast to the NIAM definition in Nijssen and Halpin [1989]. One consequence of this is that it is not possible to distinguish between "facts" that are not in a database population and "junk" that is also not in the population. It would be difficult to use the Predicator Model to discuss reasons why facts in the database population must not refer to "facts" that are not in the population.

Another consequence is that the NIAM terms Entity Type and Fact Type are not definable in the Predicator Model. The same set may model a person in one legitimate population and a car in another legitimate population. It would be difficult to use the Predicator Model to describe a business rule that the tuple ("Carol", "Physics") is to be held in the database iff the person named Carol is studying the subject named Physics.

Problem 5
The [BHW] paper declares that if a data model is structural identifiable then its structure cannot be cyclic. The justification for this declaration is specious. It is instructive to see why.

A minor flaw is that structural identifiable is defined by a formula of the form $\forall a : A \bullet \alpha$. As a result, data models with no atomic object types, such as the one in Figure 2.1.2.7 above, are deemed structural identifiable vacuously. This flaw is corrected in a later paper by changing the formula to the form $\forall x : O \bullet \alpha$.

The definition of structural identifiable uses a predicate ***Identifiable***. It is defined for fact types as well as atomic object types since there can be entities that use facts for their identification. For instance, a date could by identified by the triple (year, month, day). For a fact type f, *Identifiable*(f) is true iff *Identifiable*(Base(p)) is true for every predicator $p : f$. Of course, Base(p) can be a fact type, and this is where the trouble arises : *Identifiable* has a recursive definition.

The recursive definition determines a class, *IdSol*, (not their term) of predicates that conform to the definition. If the structure described by O and Base has the appropriate recursion theorem then *IdSol* has exactly one member, which can justifiably be called *Identifiable*.

However, when the information structure is cyclic it is possible for *IdSol* to have several members. Consider the data model in Figure 2.1.2.7 above where the fact type $f =_d \{p\}$ and Base(p) $=_d f$. Here the definition of the predicate is

$$Identifiable(f) \Leftrightarrow_d Identifiable(\mathrm{Base}(p)), \text{ which is}$$
$$Identifiable(f) \Leftrightarrow Identifiable(f)$$

IdSol has two members; call them $Identifiable_1$ and $Identifiable_2$. $Identifiable_1(f)$ is true and $Identifiable_2(f)$ is false. Both obey the definition, but in these circumstances the definition is obviously not a very useful one. Even so, it is not safe to conclude that for all data models, if it is structural identifiable then it is not cyclic.

There is obviously a problem in Figure 2.1.2.7, and it can be found by looking at populations. Suppose the population function *Pop* satisfies IsPop(*I*, *Pop*) where *I* is the information structure for this data model. Suppose the set $x_0 \in Pop(f)$. Then, by the definition of IsPop, x_0 must be a 1-tuple whose only element is some $x_1 \in Pop(f)$. As tuples are represented as families there must be a chain of memberships of the form $x_1 \in \ldots \in x_0$. Similarly, x_1 has one element $x_2 \in Pop(f)$, and so on. In general, for every i : Nat there is a set $x_i \in Pop(f)$ that is a 1-tuple whose only element is some $x_{i+1} \in Pop(f)$, with $x_{i+1} \in \ldots \in x_\mathrm{i}$.

Joining all the memberships together, we get the infinite chain

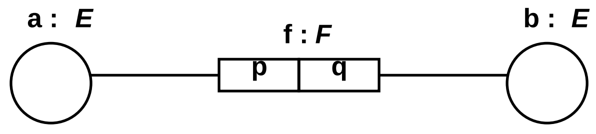
$$\ldots \in x_{i+1} \in \ldots \in x_i \in \ldots \in x_1 \in \ldots \in x_0.$$

But the axiom of foundation precludes such a chain, from which we must conclude that $x_0 \notin Pop(f)$. In other words, in any legitimate database instance $Pop(f) = \emptyset$. The same will be true for any Fact Type symbol occurring in a circuit. (The membership chain simply has more terms between the *x*'s). Such Fact Type symbols represent parts of the database that contain no information. It would therefore be reasonable to declare that "proper" data models are not cyclic. The [BHW] paper makes no mention of this emptiness property of the Predicator Model, nor does any of its successors.

Alternatively, perhaps the paper uses a set theory that lacks the axiom of foundation, but it does not say so.

Problem 6
There is another problem that can arise in the definition of *Identifiable*. Consider the data model in Figure 2.1.2.8. It has two entity types, *a* and *b*, and one fact type, *f*. Assume that there are constraints requiring that, in any legitimate population, *Pop*(*f*) defines a bijection between *Pop*(*a*) and *Pop*(*b*). In these circumstances, the definition of *Identifiable* states that $Identifiable(a) \Leftrightarrow_\mathrm{d} Identifiable(b)$; equally it states that $Identifiable(b) \Leftrightarrow_\mathrm{d} Identifiable(a)$. Once again, the definition is obviously not a very useful one and for much the same reason. The identification of an entity type depends on the identification of other object types, and it is possible for a dependency chain to form a circuit, as it does here.

Figure 2.1.2.8 A data model that has an identification problem



For identification to be possible, a second structure, that of the identification dependencies, must also be free of circuits. To complicate matters, there may be a choice of dependency structures. Identification is possible if some choice has the appropriate properties. The [BHW] paper requires identification dependencies to be chosen in a way that achieves the purpose of identification, but says nothing about any rules governing this choice. Nor does it say that the choice should be recorded in the data model so that the implemented database contents can be interpreted correctly.

Weakness 4
The Predicator Model is used to describe the desirable properties of any finished data model. It is not obvious that it can be used effectively to describe the incremental

construction of a data model, for instance in the operation of an ER diagram editor that is required to be helpful to its users.

Weakness 5
[BHW], p 476, contains two expressions, IsPop(*I*, *Pop*) and R(*I*), which imply the existence of a class of information structures of which *I* is a member. The class is unnamed, and perhaps undefined. This illustrates one of the points made in Scheurer [1994], p412, that the usual mathematical technique of defining a single instance, as in $I =_d (P, O, Sub, F)$ here, has weaknesses when modelling a class of objects.

Conclusions
Can the Predicator Model be used as a tool for answering the questions in Section 1.1?

The Predicator Model has some technical problems but these could be remedied. It describes a NIAM conceptual data model as a graph (of the nodes and edges kind) and uses a population function assigning a, possibly arbitrary, value to each node to describe a database instance.

Unfortunately, the use of population functions ignores an important NIAM principle : that a data model introduces certain fixed sets, the Entity/Label Types, of primitive objects and determines certain fixed sets, the Fact Types, of tuples. We must conclude that the Predicator Model is not a suitable vehicle for analysing what a data model does.

The graph used in the Predicator Model has a node for each Subtype symbol. But it is a NIAM principle the Subtype symbols are constraint symbols, not part of the basic structure of the data model. We must avoid the assumption that similar-looking pictorial symbols must necessarily be modelled by similar kinds of object.

2.1.3 Extensions to the Predicator Model - ter Hofstede & van der Weide [1993]; Bronts et al [1995]

References

[HW] ter Hofstede, A H M and van der Weide, Th, P [1993]. Expressiveness in conceptual data modelling.
Data and Knowledge Engineering, Vol 10, No 1, p65-100.
Note : *Four pages are blank in the British Library copy.*

[BMP] Bronts, G H W M, Brouwer, S J, Martens, C L J, and Proper, H A [1995]. A unifying object role modelling theory.
Information Systems, Vol 20, No 3, p213-235.

Highlighting

Special terms defined in the papers are highlighted in bold at the point where they are introduced.

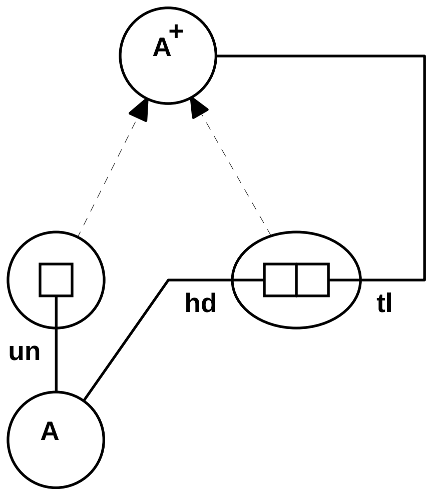
Summary

These two papers extend the Predicator Model. Although the models differ in some details and are given different names they are essentially the same. The members of database populations must now be drawn from a certain set Ω, which is defined inductively in the [BMP] paper. Four new kinds of object type symbol are introduced, as follows.

A **power type** symbol represents a power set. Each such symbol has an associated object type symbol, which can be of any kind, called its **element type** symbol. Each member of a valid population of a power set symbol must be a non-empty subset of the population of its associated element type symbol. Neither paper gives any justification for excluding the empty subset. (Later papers allow the empty subset. E.g van Bommel et al [1996]).

A **generalised object type** symbol represents a union of disparate sets. Each such symbol has one or more associated object type symbols, which can be of any kind, called its **specifiers**. A valid population of a generalised object type symbol is the union of the populations of its specifiers. This symbol can be used to describe inductively generated domains. (Only in effect; the model does not have domains as such). An example taken from the [HW] paper is shown in Figure 2.1.3.1 below. Dotted arrows lead from the specifiers to the generalised object type symbol, which is A^+ here.

Figure 2.1.3.1 An inductively generated domain



A **sequence type** symbol represents a finite sequence. Each such symbol has an associated object type symbol, which can be of any kind, called its element type symbol. Each member of a valid population of a sequence type symbol must be a non-null sequence of members of the population of its associated element type symbol. Neither paper gives any justification for excluding the null sequence. In the [HW] paper the symbol is defined to stand for the inductively generated type defined in Figure 2.1.3.1, where A^+ is the sequence type associated with the object type *A*.

A **schema type** symbol holds a data model inside itself. Each member of a valid population of a schema type symbol must be a valid population function for its internal data model. Thus an element of a tuple can be declared in a data model to be an instance of another database.

Comments

Comments on the papers have been numbered and classified as in the previous section.

Weakness 1
Each of the new object type symbols can be used to hide an identification problem. For instance, teams in a darts match might have team names, such as "Team B" or "The Magnificent Two". On the other hand, teams might have to be described by the names of their team members, as in the "Bill and Roy" team. A power type symbol would then be used to represent darts teams. It may be that the database implementation will use artificial team identifiers such as team 3 for the "Bill and Roy" team. The [HW] paper declares that in these circumstances the conceptual data model should still use the power type symbol, not a separate entity type symbol for teams, as the former describes the problem and the latter describes the solution.

There is some merit in this argument, but there is also a counter argument. The use of artificial identifiers will cost money. Certainly the business process manual must be extended and people trained accordingly. In some cases the cost can be quite high : bar code readers must be purchased; all students must be given a plastic card. Spending this money must not be an implementation decision. It must be approved as early as possible, and be justified. What better justification than a conceptual data model that clearly has an identification problem. (In the example, darts teams would have no associated labels).

Problem 1
A sequence type symbol whose element type is *A* is declared in [HW] p88-91 to be a notational shorthand for the symbol A^+ in Figure 2.1.3.1 above. This seems too implementation specific. First, a sequence such as [*a*, *b*, *c*] is specifically to be represented by the tuple $(\mathsf{hd} \mapsto a, \mathsf{tl} \mapsto (\mathsf{hd} \mapsto b, \mathsf{tl} \mapsto (\mathsf{un} \mapsto c)))$. Second, there is no way of representing the empty sequence. Third, if [*a*, *b*, *c*] is a member of a valid population then the sequences [*b*, *c*] and [*c*] must also be members.

Weakness 2
Figure 2.1.3.1 could be said to define the set

$$A^{++} =_{\mathrm{d}} (A) \cup (A{\times}A) \cup (A{\times}(A{\times}A)) \cup (A{\times}(A{\times}(A{\times}A))) \cup \ldots$$

(note : indexes have been suppressed here). Clearly, A^{++} could also be defined using an infinite set of Fact Types. In other words, the new symbols allow an infinite data model to be described by a finite data model.

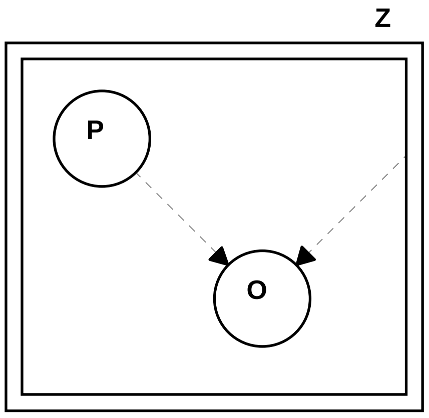
Problem 2
There is a problem with the new symbols. Practitioners might use them with too much enthusiasm and too little judgement. This is not an insignificant problem. Both papers contain examples of design errors in the use of the new symbols.

Problem 2.1
The first example of an error occurs in [HW] Figure 29 where it is claimed that any schema type symbol can be replaced by a combination of old symbols and power type symbols. The population of a schema type symbol is a set of population functions from the symbols inside the schema type to suitable sets. Each of these population functions can be represented by a tuple, belonging to a Fact Type, each of whose elements is a member of the population of a power type symbol, it is claimed. Unfortunately, a symbol assigned an empty population cannot be described this way as $\varnothing$ is not allowed in the population of a power type symbol.

Problem 2.2
The second example of an error occurs in [BMP] Figure 9, which is a data model of a collection of zoos. For each kind of animal kept by a zoo there is exactly one feeding table, represented as a sequence of times. Presumably, the reasoning is that "table" implies "list", which implies "sequence". But one possible sequence of times is (08:00, 13:00, 08:00), which is silly. A feeding table is a set of times, such as {08:00, 13:00}, so a power type symbol should have been used. Even this can be improved. A function from animal kinds to sets of times can be described as a relation from animal kinds to times without using any of the new symbols at all, and it makes queries and updates simpler.

Problem 2.3
The third example of an error also occurs in [BMP] Figure 9, but it is not so easy to detect. Zoos are represented by a schema type symbol. Each animal kept by a zoo is owned either by a person or by a zoo. This part of the data model is shown in Figure 2.1.3.2 below. *Z* is the schema type symbol representing zoos, *P* is the entity type symbol representing people, and *O* is the generalised object type symbol representing owners, alias people and zoos.

Figure 2.1.3.2 An extract from the zoo data model



Z	**Zoos**
P	**People**
O	**Owners (people and zoos)**

Any population function, *pop*, for the entire data model assigns a set of population functions to Z, each defined on the symbols within Z. Suppose Q is one of the population functions assigned to Z, so $Q \in pop(Z)$. Q will assign some population, o, to the symbol O, so $\langle O, o\rangle \in QGr$, the graph of Q. Suppose x is a member of the population of owners in the zoo Q, so $x \in o$. There is now a chain of memberships

$$x \in o \in \ldots \in \langle O, o\rangle \in QGr \in \ldots \in Q \in pop(Z)$$

From this chain we can conclude that $x \neq Q$ (axiom of foundation). In other words, Q cannot be an owner in the zoo Q; no zoo can own any of the animals it keeps! This was not the intended meaning of the data model.

Conclusions

Should the new symbols be considered when answering the questions posed in Section 1.1?

All of the new symbols described in these papers are substitutes for constructions using the basic NIAM symbols. For instance, a power type symbol is a substitute for an Entity Type whose members represent collections and a Fact Type whose tuples indicate membership of these collections. We can conclude that the new symbols can be given a lower priority than the basic NIAM symbols.

The papers have demonstrated that the symbols are difficult to use properly. We can conclude that their proper use merits careful study, but that this study can reasonably be deferred.

2.1.4 Additional papers using the Predicator Model
- ter Hofstede et al [1992]; ter Hofstede et al [1993];
- Proper & van der Weide [1994]; van Bommel et al [1994];
- Proper & van der Weide [1995]; ter Hofstede & Verhoef [1996];
- ter Hofstede et al [1996]; van Bommel [1996];
- Proper [1997]; Kovács & van Bommel [1997];
- Kovács & van Bommel [1998]

References

ter Hofstede, A H M, Proper, H A, and van der Weide, Th P [1992]. Data modelling in complex application domains.
in : Advanced information systems engineering : 4th international conference CAiSE '92, Manchester, UK, May 12-15, 1992, Proceedings, p364-377.
Springer-Verlag : Lecture Notes in Computer Science 593.

ter Hofstede, A H M, Proper, H A, and van der Weide, Th P [1993]. Formal definition of a conceptual language for the description and manipulation of information models.
Information Systems, Vol 18, No 7, p489-523.

Proper, H A and van der Weide, Th P [1994]. EVORM : A conceptual modelling technique for evolving application domains.
Data and Knowledge Engineering, Vol 12, No 3, p313-359.

van Bommel, P, Kovács Gy, and Micsik A [1994]. Transformation of database populations and operations from the conceptual to the internal level.
Information Systems, Vol 19, No 2, p175-191.

Proper, H A and van der Weide, T P [1995]. A general theory for evolving application models.
IEEE Transactions on Knowledge and Data Engineering, Vol 7, No 6, p984-996, Dec.

ter Hofstede, A H M and Verhoef, T F [1996]. Meta-CASE : Is the game worth the candle?.
Information Systems Journal, Vol 6, No 1, p41-68, Jan.

ter Hofstede, A H M, Proper, H A, and van der Weide, Th P [1996]. Query formulation as an information retrieval problem.
The Computer Journal, Vol 39, No 4, p255-274.

van Bommel, P, Frederiks, P J M, and van der Weide, Th P [1996]. Object-oriented modelling based on logbooks.
The Computer Journal, Vol 39, No 9, p793-799.

Proper, H A [1997]. Data schema design as a schema evolution process.
Data and Knowledge Engineering, Vol 22, No 2, p159-189.

Kovács, Gy and van Bommel, P [1997]. From conceptual model to OO database via intermediate specification.
Acta Cybernetica (13), 1997, p103-140.

Kovács, Gy and van Bommel, P [1998]. Conceptual modelling-based design of object-oriented databases.
Information and Software Technology, Vol 40, No 1, 1998, p1-14.

Summary

These papers are included for completeness. All of them define the Predicator Model in more or less detail, then use it to make some point.

2.1.5 Equivalent schemas
- Halpin [1991]

References

Halpin, T [1991]. A fact oriented approach to schema transformation.
in : Thalheim, B, Demetrovics, J, and Gerhardt, H-D [1991]. MFDBS91 : Proceedings, 1991 symposium on mathematical fundamentals of database and knowledge base systems, p342-356.
Springer-Verlag, Lecture Notes in Computer Science 495.

Summary

This paper presents an outline of Halpin's formalisation of NIAM conceptual data models. The full version appears in Halpin's PhD thesis. He models each data model by a first order language of predicate logic. For each Fact Type there is a predicate symbol with the same arity. For each Entity/Label Type there is a unary predicate symbol. Axioms specific to each data model declare that Entity Types are disjoint, specify constraints, etc. Axioms common to all data models define common operations on Label Types such as +, -, etc.

A database instance is modelled by additional axioms. For each fact held in the database there is an axiom which declares that objects related by certain predicates exist. For this to work properly the axioms must include constants belonging to Label Types. For instance, it is no use saying 'Some person studies some subject'; the axiom must say 'A person named "Carol" studies a subject named "Physics" '. A general axiom will have said that "Carol" uniquely identifies a person, etc.
A fact, *f*, that plays a role in another fact, *f1*, is modelled in two ways. First, *f* is modelled by an axiom, as above. Second, *f* is modelled by a tuple, explicitly represented by nested couples, in the axiom modelling *f1*. For instance, the axiom for *f* could assert $a\ R\ b$, and the axiom for *f1* could then assert $\langle a, b\rangle\ S\ c$. Axioms common to all data models declare that given any x, y then the couple $\langle x, y\rangle$ exists and has the usual properties.

There is often a choice as to how requirements are to be translated into a data model. Halpin uses his formalisation to investigate provably equivalent constructions. Several equivalence theorems are presented in this paper and in his book (see Section 2.1.1.1).

Comments

A data model and one of its database instances is described by a first order theory. Thus any results about all database instances have to be phrased as meta-theorems. Any results about all data models and their evolution or manipulation have to be phrased as meta-meta-theorems.

The model cannot declare that each Entity Type has a fixed domain. Nor can it describe illegitimate database instances as these correspond to inconsistent theories, and are effectively indistinguishable.

It is not clear how one would prove that a given fact is not present in a database instance when no constraints preclude it. This would make it difficult to prove that a database update operation inserts no more facts than it should.

The model is best used to describe completed data models that have passed all their NIAM quality checks. In particular, database instances cannot be described unless there is a satisfactory identification scheme.

Conclusions
The formalism cannot declare that Entity Types have fixed domains and is not well adapted for describing evolving data models. We must conclude that it is not a suitable tool for answering the questions in Section 1.1.

2.1.6 IFO
- Abiteboul & Hull [1987]

References

Abiteboul, S and Hull, R [1987]. IFO : A formal semantic database model.
ACM Transactions on Database Systems, Vol 12, No 4, p525-565. (December).

Summary

IFO is a style of data modelling that is quite different from NIAM. It has symbols for cartesian product, finite power set, and binary partial function (drawn as an arrow). For instance, the NIAM Fact Type described by 'Student gets Mark in Subject' can be represented in IFO by the cartesian product $Student \times Mark \times Subject$, or by the Curried partial function $Student \mathrel{+\!\!\rightarrow} (Subject \mathrel{+\!\!\rightarrow} Mark)$, or by $Subject \mathrel{+\!\!\rightarrow} (Student \mathrel{+\!\!\rightarrow} Mark)$. Only one of these descriptions is allowed in a given data model.

Note that the word "semantic" in the title means that an IFO data model can describe the database users' notions. That is, the data model is not restricted to a description of the database implementation.

An IFO data model is defined to be a directed graph constructed from several kinds of vertex and several kinds of edge. The class of all well formed IFO data models is defined in several stages, each stage introducing different kinds of vertices and edges. Some vertices, but not all, have an associated domain, for instance Nat and Persons. Some kinds of subgraph also have an associated domain, defined recursively, such as Nat×Persons and all finite subsets of Persons.

An instance of a database specified by a well formed IFO data model is defined to be an assignment of objects to each vertex and function arrow that conforms to constraints determined by the domains and the structure of the graph. The authors have proved that there is no vertex or function arrow that is forced to have an empty assignment of objects in every database instance. Thus, a well formed IFO data model is always a sensible data model.

Comments

The use of function arrows suggests that some kinds of query will be regarded as more natural than others. Perhaps unusual kinds of query will not be implemented at all.

An IFO database instance is by definition a legitimate instance. This makes it difficult to describe a complex database update as a sequence of atomic updates that may, temporarily, result in an illegitimate assignment of objects to symbols. This difficulty is the result of not defining domains for function arrows, nor for some kinds of vertex.

Conclusions

The IFO model is similar to the Predicator Model in that a data model is modelled as a graph, and a database instance is modelled as an assignment of suitable sets of objects to the vertices (and some edges). It differs in that it is defined in stages, which makes the definitions clearer (though some stages are not as clear as they should be). We can conclude that a modular development of a model is preferable.

2.1.7 HERM
- Thalheim [1993]

References

Thalheim, B [1993]. Foundations of entity-relationship modeling.
Annals of Mathematics and Artificial Intelligence, Vol 7 (1993), No 1-4, p197-256.

Summary

This paper defines HERM, alias higher-order entity-relationship model : a style of data modelling extending Chen's Entity-Relationship style (see Section 2.2.2). It includes most of the extensions proposed by others. The paper cites a criticism of Entity-Relationship data models that they lack a precise definition. (After nearly 20 years, note). The aim of the paper is to show that a "*precise, formal definition exists*" (p199).

As usual for Chen-derivatives, a HERM data model has attributes, entity types, and relationship types. Attributes can be complex. The equivalent of objectified Fact Types are permitted. Unary relationship types are permitted and are used instead of subtypes.

The class of HERM data models is defined to be an inductively generated abstract notation. For each HERM data model there is an associated graph used to describe the usual kind of ER diagram.

Each attribute has a fixed domain, defined recursively if the attribute is complex. An instance of a database specified by a HERM data model is determined by the sets of tuples associated with each entity type and relationship type. The elements of each tuple must belong to the appropriate attribute domain or tuple population.

Comments

It may be that the abstract notation is a set of well formed expressions on an abstract alphabet. However, the paper does not make this clear and some of the examples do not support this interpretation. Doubts about the core definitions make a paper more difficult to understand.

Conclusions
We must note that the foundations of any model must be defined very clearly. We can also note that a Subtype, as defined in NIAM, is equivalent to a derived unary Fact Type.

2.2 Minor

2.2.1 Data semantics
- Abrial [1974]

References

Abrial, J R [1974]. Data semantics.
in : Klimbie, J W and Koffeman, K L (Eds) [1974]. Data base management : Proceedings of the IFIP working conference on data base management. North-Holland Publishing Company, p1-60.

Summary

This paper is an early example of the definition of a data modelling notation and its meaning. A data model is defined to be an undirected graph where each node represents an evolving set of objects belonging to a particular domain, and each arc represents an evolving binary relation between the domains it connects. Parallel arcs and loops are permitted. An object can be an elementary object such as a person or an integer, or it can be a tuple. A ternary, or higher, relation R is always represented as a set T of objects representing tuples with an appropriate number of projection functions from T to R's domains. A unary relation is always represented by a binary relation to some arbitrary singleton set.

Each arc represents a binary relation, but in turn this is always represented as a pair of "access" functions. The relation $Q : X \leftrightarrow Y$ is represented by the two functions $f : X \rightarrow$ Pow(Y) and $g : Y \rightarrow$ Pow(X) determined by Q.

The construction of data models and the behaviour of the databases they describe is defined by an abstract programming language. The language is defined with the help of a data model. For instance, the construction of an arbitrary data model is described as the process of populating a database described by a fixed data model.

Comments

An object must be inserted into the database before it can be related to another object. The person Tommy must be inserted first if one wants to say that Tommy is 9 years old. The paper does not say how users should interpret the existence of an object in the database. Does it simply mean that a precondition is satisfied, or does it have a deeper significance? Note that NIAM avoids this conundrum by insisting that only facts can be inserted into a database. 'Tommy is 9' is a fact, 'We are aware of Tommy' is a (unary) fact, but 'Tommy' is not a fact.

The definition of the abstract programming language has to include the definition of scopes and of *for* loops. The definition of data models and their meaning should not be complicated by such things.

2.2.2 Chen-style ER data models; CASE tools - Chen [1976], Batini, Ceri, & Navathe [1992]

References

Chen, P P [1976]. The Entity-Relationship model - Toward a unified view of data.
ACM Transactions on Database Systems, Vol 1, No 1, p9-36.

Batini, C, Ceri, S, and Navathe, S B [1992]. Conceptual database design : An Entity-Relationship approach.
Benjamin Cummings.
Library reference : Joule 001.64/BAT.

Highlighting

Special terms defined in the paper and book are highlighted in bold at the point where they are introduced.

Summary

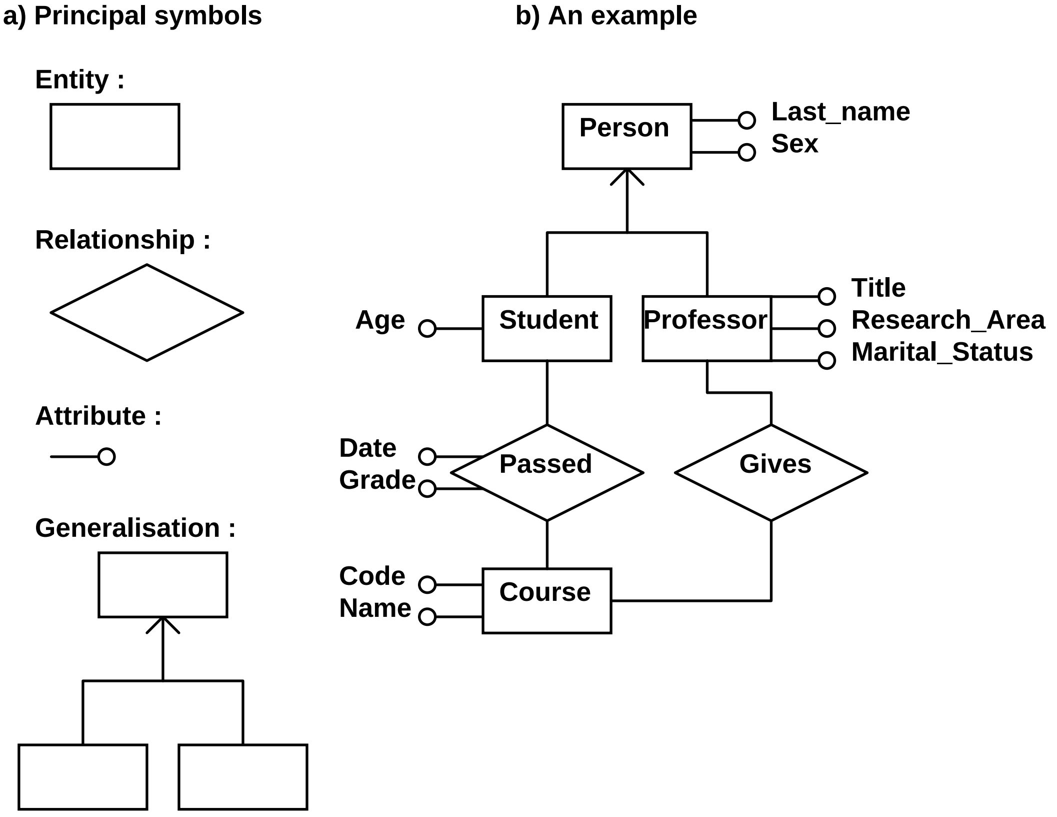
The Chen paper is an early, and much cited, paper that declares that databases can be described by conceptual data models. It describes a formal model of data models and a corresponding pictorial notation. There are many variants of the Chen style of data model. One common variant is described in the Batini et al book. The book is a textbook describing the notation and how to use it to design a database. The book finishes with a chapter discussing CASE tools relevant to database designers.

The Chen variant will be described first. A data model contains a family of **entity sets**. Each is an evolving set of entities, such as a set of employees or cars. Entity sets may overlap. It also contains a family of **relationship sets**. Each is an evolving set of tuples forming the graph of an evolving relation. The relation's (evolving) domains are elements of the family of entity sets. Finally, a data model also contains a family of **attributes**. Each attribute is defined to be an evolving function from an entity set or relationship set to a **value set** or to a cartesian product of value sets. Each value set is a fixed set of values such as a set of numbers or a set of character strings. It is not clear whether an attribute can be a partial function or must be total. Note that Chen's purpose was more to describe a more meaningful view of an evolving database than to describe a database specification. In fact, the formal description is of a single generic database instance.

In the pictorial notation each entity set and relationship set is represented by a symbol. Lines join the symbol for a relationship set to the symbols for the entity sets that form its domains. Attributes are described in text, but not shown in the diagram.

The Batini et al account differs a little from Chen's. Attributes are now shown in the diagrams and can be relations, not just functions. Entity sets that overlap are now collected into subtype/supertype structures. See Figure 2.2.2.1 below. The notation includes a way, not shown in the figure, of saying whether or not an attribute is functional, partial, etc.

Figure 2.2.2.1 Notation



The book also changes some of the terminology. An **entity** is said to represent a class of real-world objects (p31). An individual object is called an **entity instance**. However, some of the detailed definitions use "entity" to mean what Chen calls an entity set, namely the set of those objects of a certain class currently mentioned in the database (p20, 34, 25).

The last chapter of the Batini et al book discusses database design tools and describes some examples (p411-454). Two requirements for these tools are said to be :

"*Tools should also be oriented toward the underlying semantics of evolving designs, rather than the more superficial layer of graphical presentation.*" (p416)

"*A minimal requirement of an ER diagram editor is that it must eventually produce syntactically valid ER diagrams (although intermediate steps may sometimes have unconnected relationships on the screen).*" (p420)

The chapter gives the impression that even in 1991 the number of commercial products meeting these requirements was few to non-existent.

Comments

The distinction between entities and values is always treated as being universal and intuitively obvious, yet the definitions given always fail to separate the two classes. It is not right that a student should fail an exam for saying that the number 3 is a " *'thing' which can be distinctly identified*", instead of something that can be " *obtained by* *observation or measurement.*" (Both quotations come from Chen's paper. He gives 3 as an example of the latter.)

It is forbidden to describe a relation between value sets. Thus a book of logarithms cannot be described as a relation on a set of values. An artificial entity set must be introduced to separate the value set from the relationship set. This is likely to make the description unclear.

Before drawing a Chen-style ER diagram, the designer must decide which objects of interest are important (shown as entities) and which are not important (represented by attribute values). The diagram shows attribute symbols clustered round entity symbols, and looks suspiciously like the first draft of a relational database table schema. In other words, Chen-style ER diagrams appear to include more implementation decisions than conceptual data models ought to do.

2.2.3 The "UMIST" notation
- Layzell & Loucopoulos [1989]; Loucopoulos [1993];
- Theodoulidis et al [1992]

References

Layzell, P and Loucopoulos, P [1989]. Systems Analysis and Development (3rd Ed). Chartwell-Bratt.

Loucopoulos, P [1993]. Unpublished MSc course notes. (Conceptual modelling - The data perspective).

Theodoulidis, C, Wangler, B, and Loucopoulos, P [1992]. The Entity-Relationship-Time model. In : Loucopoulos, P and Zicari, R (Eds) [1992]. Conceptual modelling, databases, and CASE : An integrated view of information systems development. John Wiley & Sons, Inc, p87-115.
Library reference : Joule 001.6442/LOU.

Summary

The first two publications describe a variant of NIAM used at UMIST. It is sufficiently close to NIAM to be called NIAM-style. One difference from NIAM is that different symbols are used in the graphical notation, see Figure 2.2.3.1 and 2.2.3.2 below. Ellipses are replaced by rectangles and role names are placed outside role boxes. The other difference is that there is much less emphasis on marking constraints on diagrams, though all known constraints must be written down somewhere, of course. One of the few kinds of constraint symbol is of a kind not used in NIAM.

The Theodoulidis et al [1992] paper describes a different style of data modelling that can be used for the design of temporal databases. It has been mentioned here as its graphical notation it almost the same as that described in Figure 2.2.3.1.

Figure 2.2.3.1 Principal symbols in the notation

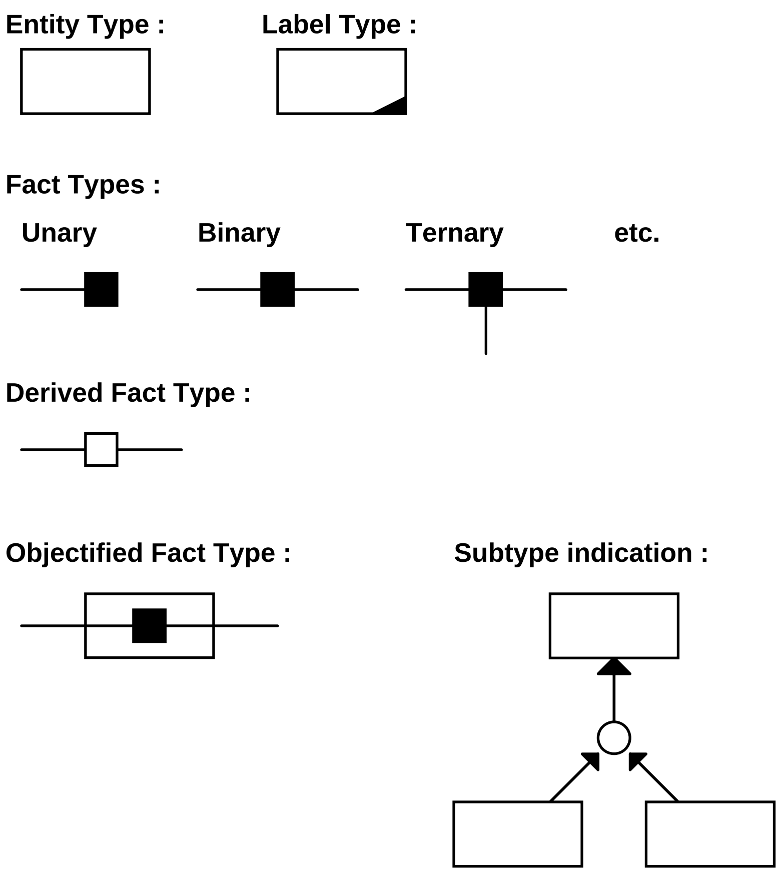
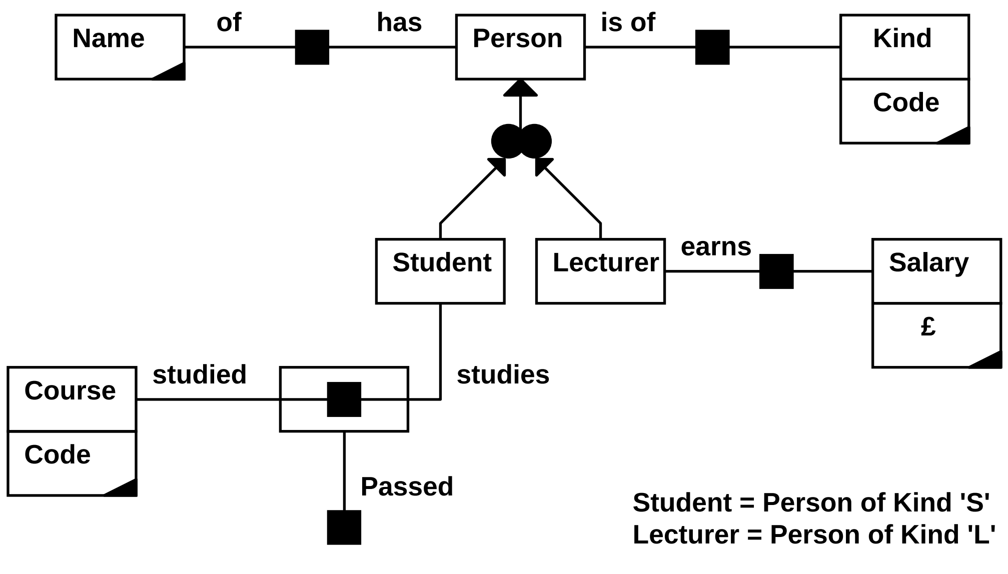


Figure 2.2.3.2 An example of the notation



Comments

The graphical symbols are much easier to draw neatly with a ruler or word processor, and long role names no longer have to be squashed into small role boxes.

2.2.4 Subtype & Supertype structure - Simovici & Stefanescu [1989]

References

Simovici, D A and Stefanescu, D C [1989]. Formal semantics for database schemas. Information Systems, Vol 14, No 1, p65-77.

Summary

The title of this paper is somewhat misleading. "Schema" means subtype supertype, alias specialisation generalisation, structures. "Semantics" means axioms defining well formed structures.

A schema is defined to be a set *S* with two, extended to three, relations on *S*. Five, extended to eight, axioms constrain these relations. *S* can be thought of as the index set of a family where relations on *S* describe properties of the family, or it can be thought of as a set of symbols in a structure diagram where relations on *S* describe lines joining the symbols.

The authors prove that the extended structures, those with three relations and eight axioms, can be inductively generated, but they do not state a set of generators. They go on to investigate the preconditions necessary to allow a large schema to be constructed by merging many small schemas. The investigation is restricted to the structures with two relations. Treatment of extended structures is flagged as belonging to future research.

Comments

The axioms are justified by showing that they have the expected consequences. It might have been simpler to define an inductively generated set of structures, then investigate their properties. The generators might have been easier to justify and the axioms would appear as theorems.

2.2.5 Z model of conceptual data models - Misic et al [1992]

References

Misic, V, Velasevic, D, and Lazarevic, B [1992]. Formal specification of a data dictionary for an extended ER data model. The Computer Journal, Vol 35, No 6, p611-622.

Summary

This paper presents a **Z** model of a variant of Chen-style conceptual data models. The variant includes subtype/supertype structures and the equivalent of objectified Fact Types.

The entity set symbols and relationship set symbols in a data model and the lines joining them are modelled as a directed acyclic graph, with parallel arcs permitted.

Subtype/supertype structures are modelled as a directed acyclic hypergraph on the entity set and relationship set nodes. Attributes are modelled as an injective relation from the entity set and relationship set nodes to nodes modelling value set symbols.

Comments

The paper illustrates a common problem in **Z** models. **Z** uses many-sorted set theory. If the union, etc, of two sets is needed then they must belong to the same sort. (Which is why Nat is defined to be a subset of Integer, but not of Real). Thus the model starts by defining a "catchall" set CONCEPTS. Most sets in the model are defined to be subsets of CONCEPTS.

The **Z** schemas in this paper are rather untidy. They do not communicate the specification as readily as they ought to.

2.2.6 Category description of data models - ter Hofstede et al [1996], Frederiks et al [1997]

References

ter Hofstede, A H M, Lippe, E, and Frederiks, P J M [1996]. Conceptual data modelling from a categorical perspective.
The Computer Journal, Vol 39, No 3, p215-231.

Frederiks, P J M, ter Hofstede, A H M, Lippe E [1997]. A unifying framework for conceptual data modelling concepts.
Information and Software Technology, Vol 39, No 1, p15-25.

Summary

The objective of the ter Hofstede, et al, paper is stated very clearly in its abstract : "*In depth comparison of* [conceptual data modelling] *techniques are very difficult as the mathematical formalizations of these techniques, if they exist at all, are very different. Consequently, there is a need for a unifying formal framework providing a sufficiently high level of abstraction. In this paper the use of category theory for this purpose is addressed.*"

In outline, the generic model defined in the paper is as follows. A data model is declared to be a directed edge-labelled graph, *G*, with certain restrictions on its structure. Each candidate database instance, alias population, is described by some image, via a graph homomorphism, of *G* in a category *C*. Appropriate objects of *C*, determined by *G* and its edge labelling, are declared to be the components of the database instance. To be a proper database instance the image must satisfy certain constraints, defined in the usual way by saying that certain diagrams in *C* must commute.

The semantics of *G* are partly fixed by the rules of the generic model, and partly variable by the choice of the category *C* and by any additional constraints on proper images. For instance, *C* could be the category Set, where arrows represent total functions, or the category Rel where arrows represent possibly partial relations. For NIAM data models the category is Set and there are some additional general constraints.

The Frederiks, et al, paper is an introduction to the first paper.

Comments

The ter Hofstede, et al, paper starts by saying that category theory has been chosen so that representation issues can be ignored, but then makes several specific representation decisions. One consequence is that Chen-style ER diagrams cannot be modelled directly; they must be translated to fit the requirements of the graphs G.

Category theory is concerned with structure, whereas conceptual data models are also concerned with equality and membership. It is not surprising that the generic model has no construct corresponding to fixed domains, and that it allows a data model to ask for a set that is a member of itself.

The very generality of category theory has resulted in a generic model that says as much about the many ways that a database can be implemented as it does about the common semantics of those implementations. For instance, data can be replicated; tuples can be implemented as tuples of pointers.

It becomes obvious on reading the papers that before two styles of data modelling can be compared using categories the syntax and semantics of each style must be well established, using whatever formalism is appropriate.

2.3 Ancillary

2.3.1 Databases

2.3.1.1 Databases look like sets of relations
- Codd [1970]

References

Codd, E F [1970]. A relational model of data for large shared data banks.
Communications of the ACM, Vol 13, No 6, p377-387, (June 1970).

Summary

This paper is an early, and much cited, paper that declares that **any** database can be represented as a family of evolving n-ary relations ($n \in \{1, 2, 3, \ldots\}$). It also points out that application software and users will be insulated from most database reorganisations and enhancements if their interface to the database uses this representation.

The relations all have simple domains, such as numbers or strings. This simplifies the application interface, and the applications. The paper points out that any relation with a domain that is a set of relations can be, and should be, recast as one or more relations with simple domains.

Comments

Note that the tables implemented by most relational database management systems can hold blank values, alias nulls. The application interface presents tables rather than the relations advocated by Codd. A simple way to describe the difference is to say that for pragmatic reasons one table may hold the graphs of several relations in an encoded form, with the consequence that nulls are possible.

2.3.1.2 Data and reality
- Kent [1978]

References

Kent, W [1978]. Data and reality : Basic assumptions in data processing reconsidered.
North-Holland Publishing Company.
Library reference : Rylands 510.8/K19.

Summary

This book describes the problems and misunderstandings that any database designer must always be on the lookout for. For instance, is a "manufacturing item" something that has a quantity, as for nuts and bolts, or something that has a serial number, as for railway engines?

The book is said to have been an early influence in the evolution of NIAM.

2.3.1.3 Theory of relational databases - Maier [1983]; Date [1995]

References

Maier, D [1983]. The theory of relational databases.
Pitman Publishing Ltd.
Library reference : Joule 001.64/MAI.

Date, C J [1995]. An introduction to database systems (6th Ed).
Addison-Wesley Publishing Company.
Library reference : Joule 001.6425/DAT.

Summary

These two textbooks describe the theory and practice of databases when represented as a family of evolving relations. Note that the books concentrate on database implementation and manipulation. There is little or no discussion of what the contents of a database might mean to its users.

The first sentence of Date's book says "*A database system is essentially nothing more than a computerized record-keeping system.*" This should remind us that databases have been in use for thousands of years. Computers are simply a convenient means to implement them.

2.3.1.4 Knowledge bases - Sowa [1991]; Brachman, et al [1991]

References

Sowa, J F (Ed) [1991]. Principles of semantic networks : Explorations in the representation of knowledge.
Morgan Kaufmann Publishers, Inc, San Mateo, California.
Library reference Rylands 501.5 S7.

Brachman, R J, McGuinness, D L, Patel-Schneider, P F, and Resnick, L A [1991]. Living with CLASSIC : When and how to use a KL-ONE-like language.
in : Sowa [1991], p401-456. (*see above*)

Summary

Both Knowledge Representation Systems (KRSs) and Data Base Management Systems (DBMSs) store information of some kind, yet the literature on the two subjects is almost entirely disjoint. Presumably there is some difference, perhaps only qualitative, that separates the two kinds of system.

The Sowa book contains a selection of papers on the theory and practice of KRSs. One of them is the paper by Brachman et al. The paper describes a typical KRS, called CLASSIC, and gives advice on when to use it and when not to use it, with guidelines on how to design an application.

The difference between a KRS and a DBMS can be illustrated by the following two imaginary conversations.

A KRS interaction :-

User	"Kermit is a frog. Tell me about Kermit."
KRS	"Kermit is a frog, an amphibian, an animal, and a thing." "Kermit is probably green." "Kermit has four legs, two eyes, one head, and no tail."

A DBMS interaction :-

User	"Kermit is a frog. Tell me about Kermit."
DBMS	"Insufficient data."
User	"Tell me about 'Kermit' playing the role 'individual'."
DBMS	"Kermit is a frog."
User	"Tell me about 'frog' playing the role 'subclass'."
DBMS	"A frog is a frog, an amphibian, an animal, and a thing."

A KRS is expected to hold information about the meaning of some of the information it can hold, and is expected to use that information. A DBMS is expected to hold information in a way that allows it to be retrieved, but the meaning of that information and the significance of its presence in the database is irrelevant to the DBMS.

Comments

In the preface to the book Sowa says "*Many of the issues that* [database] *developers are encountering are ones that have long been addressed in the AI research on semantic networks.*" Unfortunately, he makes no mention of any possibility of ideas flowing in the opposite direction.

2.3.2 Mathematical topics

2.3.2.1 Mathematical method - Russell [1919]

References

Russell, B [1919]. Introduction to mathematical philosophy.
Routledge, (republished in 1993).
This publication has an introduction by J G Slater.

Summary

This book describes the foundations of set theory as it was circa 1918. It is included here as an example of clear writing on a difficult subject, and for two quotations that apply to any form of modelling :

"*Here, as constantly elsewhere, generality from the first, though it may require more thought at the start, will be found in the long run to economise thought and increase logical power.*" (p26-27).

"*The method of 'postulating' what we want has many advantages; they are the same as the advantages of theft over honest toil.*" (p71).

2.3.2.2 Category theory - Mac Lane [1971]

References

Mac Lane, S [1971]. Categories for the working mathematician.
Springer-Verlag New York Inc.
Library reference : Rylands 512.85/M1.

Summary

This book defines the axioms and standard constructions of category theory. It is a general textbook, not specific to any particular application area.

2.3.2.3 Set theory - Enderton [1977]; Hamilton [1982]

References

Enderton, H B [1972]. A mathematical introduction to logic.
Academic Press Inc.

Enderton, H B [1977]. Elements of set Theory.
Academic Press Inc.

Hamilton, A G [1982]. Numbers, sets and axioms.
Cambridge University Press.

Summary

Enderton [1977] is an excellent and thorough treatment of the foundations of axiomatic set theory (ZF variant). In particular, it contains the definition of Well Founded relations and a statement and proof of the transfinite recursion theorem for Well Founded relations (p241-246). The theorem does not require codomains to be fixed in advance.

Hamilton [1982] covers much the same ground and includes a full statement of the VNB set axioms.

Enderton [1972] p22-25 contains a brief account of structural induction.

2.3.2.4 Feature notation
- Scheurer [1994]

References

Scheurer, T [1994]. Foundations of Computing : System development with set theory and logic. Addison-Wesley.

Summary

This book covers three topics : the basics of set theory up to structural induction and recursion theorems; propositional and first order predicate logic including the formal definition of syntax and semantics, and including deduction sequences and metatheorems; and Feature Notation, which is a flexible and precise notation for describing classes of system models constructed from sets.

2.3.3 Other topics

2.3.3.1 Definitional systems
- Geoffrion [1987]; Geoffrion [1989]
- Schubert [1991]

References

Geoffrion, A M [1987]. An introduction to structured modeling.
Management Science, Vol 33, No 5, p547-588. (May).

Geoffrion, A M [1989]. The formal aspects of structured modeling.
Operations Research, Vol 37, No 1, p30-51. (Jan-Feb).

Schubert, L K [1991]. Semantic nets are in the eye of the beholder.
in : Sowa, J F (Ed) [1991]. Principles of semantic networks : Explorations in the representation of knowledge, p95-107.
Morgan Kaufmann Publishers, Inc, San Mateo, California.

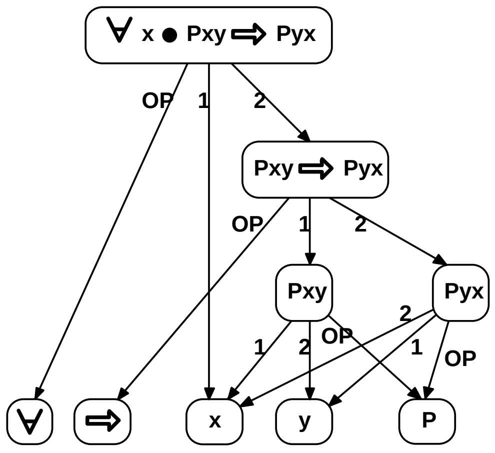
Summary

Geoffrion's concern is much like Codd's. He points out that there is a large class of operations research problems that can be described in a standard way. He advocates the use of this description in computerised modelling systems for the interface between problem declarations and problem solvers such as optimisers. This would make it easier to define problems, to solve them, to vary them, to re-use them, and also to describe them to non-experts such as the managers who need the results and need to know what kind of problem is being solved.

In Geoffrion [1989] he points out that these problem descriptions belong to an even larger class he calls **definitional systems**. The distinguishing feature of these systems is that each defines a set of objects where each object is either a simple object, which is undefined or defined elsewhere, or a complex object constructed from simple objects and less complex objects. An example, adapted from Schubert [1991], is shown in

Figure 2.3.3.1.1. Its purpose is to define a single well formed expression in a first order language (using a network as it happens).

Figure 2.3.3.1.1 A Wfe defined by a network



Comments

Even if the main purpose of the network in Figure 2.3.3.1.1 is to define one Wfe it does in fact define four that are complex and two, *x* and *y*, that are simple. Obviously, the network could be extended to include the definition of several more, such as '$\exists x \bullet Pxy \Rightarrow Pyx$', '$Pxx \Rightarrow Pyy$' and 'z'.

If the simple nodes in Figure 2.3.3.1.1 were transformed into Entity Type symbols, the complex nodes into Fact Type symbols, and the arc labels treated as role names, then the result would be a NIAM ER diagram. This suggests that conceptual data models might also belong to the class of definitional systems.

2.3.3.2 - Heuristics for problem solving Polya [1990]

References

Polya, G [1990]. How to solve it : A new aspect of mathematical method (2nd Ed). Penguin Books.

Summary

As the title suggests, this book provides heuristics for problem solving, particularly maths problems. The book starts with a two-page statement of the heuristics, on p xxxvi-xxxvii.

Comments

It would be useful to supplement the heuristics with suggestions particular to system modelling.

2.3.3.3 Methods need formalising
- Eick & Raupp [1991]; ter Hofstede & van der Weide [1992];
- McGinnes [1994]; Paynter [1995]

References

Eick, C F and Raupp, T [1991]. Towards a formal semantics and inference rules for conceptual data models.
Data and Knowledge Engineering, Vol 6, No 4, p297-317.

ter Hofstede, A H M and van der Weide, T P [1992]. Formalization of techniques : Chopping down the methodology jungle.
Information and Software Technology, Vol 34, No 1, p57-65.

McGinnes, S [1994]. CASE support for collaborative modelling : re-engineering conceptual modelling techniques to exploit the potential of CASE tools.
Software Engineering Journal, Vol 9, No 4, p183-189. (IEE, July).

Paynter, S [1995]. Structuring the semantic definitions of graphical design notations.
Software Engineering Journal, Vol 10, No 3, p105-115. (IEE, May).

Summary

Two themes appear in these papers. One theme is that most notations in common use for designing information systems lack a precise definition of their syntax and semantics. The other theme is that most CASE tools in common use fail to prevent improper use of the notation and seldom facilitate design changes.

The ter Hofstede & van der Weide [1992] paper is dedicated to these two themes and illustrates them with NIAM and the Predicator model, (not surprisingly). The paper rightly points out that it is difficult for a CASE tool to be helpful to its users if there is no general agreement on the proper use of the notation it implements.

3 Context

Some foundations must be laid before any attempt is made to answer the questions in Section 1.1.

First, the questions implicitly talk of all possible databases. We must know how to recognise a database. Presumably a NIAM conceptual data model specifies essential characteristics of a database. We must know what the essential characteristics are. Strangely, there does not appear to be a textbook or paper that provides all of this information, so it must be provided here (Section 3.1).

Second, we must understand what a NIAM conceptual data model purports to do, and the mechanisms it uses to do it. We should also know of any obvious limitations. Again, there does not appear to be a suitable textbook or paper so the information is provided here (Section 3.2).

Third, this work makes much use of Scheurer's Feature Notation. Readers unfamiliar with the notation may find a brief introduction useful (Section 3.3).

3.1 What is a database?

If we wish to choose a technique for designing a new product we need to know what kind of product it is. To make best use of known techniques we need to recognise the kind even in unfamiliar circumstances. This section illustrates the diversity of objects that can plausibly be called databases. It points out the few characteristics that are essential in any database, and some that are not.

If we wish to design a product we must have some way of describing it. We must be able to describe its component parts and their inter-relationships. We introduce a way of describing any database. For obvious reasons it is a way that can be used when discussing NIAM conceptual data models and the databases they specify.

As the final topic in this section we build a set-theoretical model of any database at any given moment. The model captures the essential characteristics of any object that can be classified as a database. It can be used as an aid to recognising databases, and as an adjunct to any model of conceptual data models. The model is somewhat unconventional so we compare it with some other, more familiar, models.

Many of the key ideas have been adapted from the work of other authors : credits are given in the final sub-section. Several terms are used here and later on with special meanings. They are highlighted in bold at the point where they are introduced.

3.1.1 Some examples

The first sentence of Date's book on databases says :

> "*A database system is essentially nothing more than a computerized record-keeping system.*" (Date [1995], p2)

Date is appealing to his readers' common knowledge but says little or nothing about what that knowledge might be. Our task here is to decide how a "record-keeping system" is to be recognised. As any database is said to be essentially a kind of record-keeping system we will use the word database for both, ignoring any distinction between computerised and non-computerised systems.

Consider the following examples. Are there any significant differences between them?

Example 1
A corporate database, containing details of a company's purchases, sales, stocks, employees, salaries, etc, etc.

Example 2
The Inland Revenue database, containing details of people and organisations, and the tax they owe and the tax they have paid.

Example 3
A company's account books, used to determine the company's profit (or loss) and its current assets and capital.

Example 4
A telephone directory, containing names, addresses, and telephone numbers.

Example 5
A dictionary, containing words and their meanings.

Example 6
An electoral register, containing a list of voters' names and their addresses.

Example 7
A railway timetable, containing the names of railway stations and the times of trains between stations.

Example 8
A dentist's appointment book, containing people's names and their attendance times.

Example 9
A book of logarithms, containing numbers and their logarithm, antilogarithm, sine, cosine, etc; also containing interpolation tables so that the logarithm, etc, of numbers not in the book can be obtained.

Example 10
A calculator, containing facilities for entering a number and obtaining its logarithm, antilogarithm, sine, cosine, etc.

Example 11
A personal diary, containing dates and their day of week, phase of moon, appointments, travel expenses, etc.

Example 12
A shopping list, containing reminders of things to be bought, showrooms to be visited, etc.

Example 13
A kitchen notice board, containing a list of things to be done, who is to do it, exhortations, quotations, general remarks about life, etc.

Example 14
Railway tickets, each containing two station names, a date, and other pertinent information.

Example 15
A museum, containing rooms and showcases holding objects that are associated in some way.

Example 16
Tags tied to museum exhibits; labels stuck to folders.

Example 17
A handkerchief with a knot tied in it, reminding the owner that something is to be remembered.

Example 18
A data dictionary, containing information about the data types and the files and tables in an information system.

There is obviously a difference of scale in the examples. A corporate database (Ex 1) could easily contain several gigabytes of data, whereas a shopping list (Ex 12) is unlikely to contain more than a few tens of words. However, a corporate database could easily have grown from a few bytes, and a shopping list could conceivably grow very much bigger. There is no obvious size above which an object can be a database and below which it cannot be a database.

Point 1
Any plausible description or definition of databases will be insensitive to scale.

A table of logarithms (Ex 9) can be described as a sample of a continuous function of the Reals. A calculator (Ex 10) can be described as a computer with an evolving execution state. These descriptions are natural and appear to have nothing to do with databases. However, a physicist might describe a corporate database (Ex 1) as an evolving wave function that is a continuous function of time. It seems unlikely that there is any object that is a database and has no alternative description whatsoever.

Point 2
An object is not precluded from being a database simply because it has other viable descriptions.

One theme common to all the examples is that each is an object that people consult. They consult it when they wish to be reminded of things they might not remember correctly without assistance. Some information, such as exhortations (Ex 13) and museum exhibits (Ex 15), are more naturally described as communicating new knowledge rather than reminders. However, this is equally true of anything a person sees for the first time in a corporate database (Ex 1). Exhortations can still be re-read and museums re-visited if a reminder is needed.

Point 3
A **database** is a kind of external memory.

Note that there is no inherent restriction on the things that people might wish to be reminded of. They might even wish to be reminded of database design details (Ex 18).

The contents of a corporate database (Ex 1) will usually be changing, possibly many times a second. Logarithms do not change : $\log_{10} 2$ is fixed for all time by its very nature. However, anyone who wishes to be reminded of the value of $\log_{10} 2$ will consult a book of logarithms (Ex 9) or a calculator (Ex 10). The object they consult came into existence at some time and will be replaced in a few years by a new version that, perhaps, is more useful or has fewer errors. The contents of the objects consulted do change, but much more slowly than the contents of a corporate database.

Note that books, calculators, and so on are described here as being individual **instances** of an evolving object. Unfortunately, the same words are often used to refer to both an evolving object and its instances. For example, a railway timetable (Ex 7) could be a fixed instance (giving today's train times) or an evolving publication (which changes every six months). The distinction is vital, but seldom made clear.

Point 4

a) A database's contents evolve with time. The speed of this evolution is not significant on its own.

b) It is necessary to distinguish between evolving objects and their instances.

A corporate database (Ex 1), a telephone directory (Ex 4), an electoral register (Ex 6), and several of the other examples are each restricted to holding information on certain topics. There are rules that constrain their contents. For instance, a telephone directory must not say who is an elector; an electoral register must not include telephone numbers. A corporate database might say both, but might not be allowed to hold details of directors' share holdings.

On the other hand, a personal diary (Ex 11), a shopping list (Ex 12), a kitchen notice board (Ex 13), and several of the other examples are not restricted at all. The users have a free choice of topics. Even so, it may be convenient to say that their contents are also constrained, but that the constraint rule is a loose one that precludes no topics at all. Alternatively, it could be said that the constraint rules also evolve, almost as fast as the contents evolve.

Point 5
Typically, the possible contents of a database are constrained by some rules. This can be said to be universally true if vacuous constraints are permitted.

A book of logarithms (Ex 9) may contain a table giving $\log_{10} 2.14$ directly, but require the use of interpolation tables to obtain $\log_{10} 2.1497$. Nevertheless, it is reasonable to say that the book contains both of these logarithms. In the case of a calculator (Ex 10), the user will not know how a logarithm was obtained. Similarly, a corporate database (Ex 1) might not contain a direct statement of sales so far this year, but the user might still be able to deduce the amount. Again, it is reasonable to say that the database contains this information. In effect, some of the information in a database can be held in an encoded form. This might or might not be ascertainable by the people who consult the database. There is no obvious way of measuring the degree of encoding, and no obvious limit on how high the degree of encoding can be.

Point 6
Information in a database can be held directly or indirectly. This is not significant to the database users, and might not be observable by them.

In all of the examples there is information held in some kind of recording medium, to give it persistence. Which kind of recording medium is mostly a matter of convenience and available technology. For instance, railway timetables are recorded in computers, on paper, and on blackboards. A museum exhibit such as a Greek vase is an exceptional case. Here, there is still a recording medium but it is the vase itself.

Point 7
Every database has a recording medium, but the kind of medium is not significant. Some possible recording mediums are :

A network of computers
A computer
Sheets of paper or card
A whiteboard
Tally sticks
Clay tablets
A museum display cabinet
A person with a trustworthy memory
A well-trained parrot

There are further points to be made, but they are best illustrated by the case study in the next section.

3.1.2 Case study : the Plunder Inn register

Example 19
This example is taken from a C++ programming exercise used in the MSc Computation conversion course at UMIST. The essentials of the program's specification are :

"*Plunder Inn is a shelter for pirates. It maintains a register recording for each resident his/her name, date of arrival and date of departure (only one stay at most per pirate is recorded). You are … to represent this register as an array of records.*" "*… the program will prompt the user to input a name, a date of arrival and a length of stay.*" "*The length of stay will be a nonnegative integer number of days.*"

The format of individual records in the register is specified :

```
struct Resident
  { NAME name;
    DATE arrival;
    int  LenStay;
    DATE departure;
      // Invariant: arrival plus LenStay = departure
  };
```

(The types NAME and DATE are also specified).

Two points can be made immediately.

Point 8
Even entirely imaginary requirements or customers can give rise to implemented, working, databases.

Point 9
One statement of requirements can give rise to many concurrent implementations (one per student here). Each implementation evolves independently but to the same rules (programming errors and omissions excepted, of course).

Pretend for the moment that the requirements are real. The owner of Plunder Inn maintains a register containing names, dates, and numbers. But it is not names, alias character strings, that occupy rooms, owe money, and can be sued. It is pirates. Equally, it is not dates and numbers that rooms are occupied for and pirates are charged for, but days.

When the owner of Plunder Inn consults the database via the program he might be reminded of the following **information items** :

Example 19.1
'A particular pirate arrived on a particular day';
'that pirate has the name "Jack" ';
'that day has the date "1.11.96" ';
'the name "Jack" is associated with the arrival date "1.11.96" '.

Of course, the first three items are held indirectly in the database, an extension of Point 6. Even the last item is held indirectly. First, the text "the name" and "is associated with the arrival date" is not stored by the program. Second, "Jack" and "1.11.96" had to be extracted from a record containing two other parts.

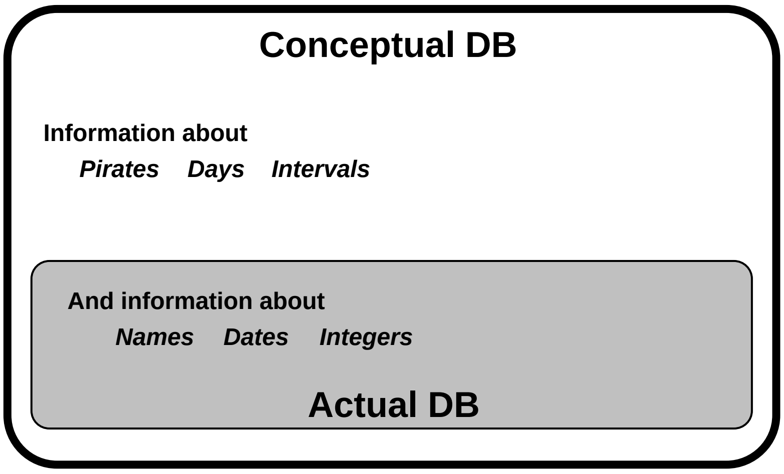
For any database, and its instances, it is likely that the users can describe its contents in two different ways. First, they can describe it in terms of the things that matter to them. Here, this is information concerning pirates, days, time intervals, names, dates, and numbers. Second, they can describe it in terms of things that are immediately visible to them in the database. Here, this is only names, dates, and numbers. A casual observer of the Plunder Inn program would use only the second description, but the owner of Plunder Inn must know and understand the first description if the program is to be of any use.

It is convenient to have names for these two descriptions.

Point 10
Any database has at least two descriptions :
a) as a **conceptual database**,
holding information about things that matter to the owners and users;
b) as an **actual database**,
holding information that is immediately visible to users.
By definition, the actual database is nested inside the conceptual database : any information in the former is also in the latter. See Figure 3.1.2.1 below.

Another example highlights the difference between these two descriptions. In a dentist's appointment book (Ex 8) the only things that are visible to readers are text strings (actual database), but the dentist will say that the book contains appointments, identified with the help of text strings (conceptual database).

Figure 3.1.2.1 Two descriptions of the Plunder Inn register



It is important to recognise that there is no absolute definition of the conceptual database. For instance, the owner of Plunder Inn can deduce the number of rooms occupied on certain days. If this is important to him then this information is explicitly declared to be in the conceptual database; if it is not important, then it need not be. Although unimportant information may be deducible, the database implementation is under no obligation to make it easy to deduce. Nor is there any obligation to mention it in any description of the database.

Point 11
The extent of the information declared to be in a conceptual database is a design decision; it has no absolute definition.

Equally, there is no absolute definition of the boundary between a conceptual database and an actual database. For instance, the owner of Plunder Inn requires that the departure date corresponding to a recorded arrival date and length of stay is immediately visible to users of the database. Information on departure dates clearly resides in the actual database even if they are calculated on demand but not stored in records. A database holding information about digital pictures might store the pictures themselves in the database. They would then be held in the actual database. Alternatively, they might be stored separately outside the database. They would then not be held in the actual database even if they are described as being in the conceptual database. The choice would depend on storage requirements, copyright restrictions, user needs, etc.

Point 12
The placement of the boundary of an actual database within a conceptual database is a design decision; it has no absolute definition.

There is a small problem with conceptual databases that should be mentioned here. Suppose the Plunder Inn register is maliciously falsified to say
' The pirate "R2D2" arrived on "1.11.96" for "3" days, departing on "4.11.96" '.
We happen to know that R2D2 is not a pirate, so how can we describe the contents of an instance of the conceptual database if it contains this kind of entry? Perhaps we should talk to the owner of virtual pirates who exist and do not exist at the same time. Or, we could talk of temporary or honorary pirates. Note that there is no such problem with the actual database. Anything visible to users obviously exists, however silly.

Point 13
Acting as an external memory, a database can contain false memories, even illogical ones.

There are more ways of describing any database. One is what one could call the implementer's description. The C++ declaration of the Resident structure in Example 19 belongs to this kind of description. Another is a physical description, saying how information is encoded in the recording medium itself. Again, it is convenient to have names for these two descriptions.

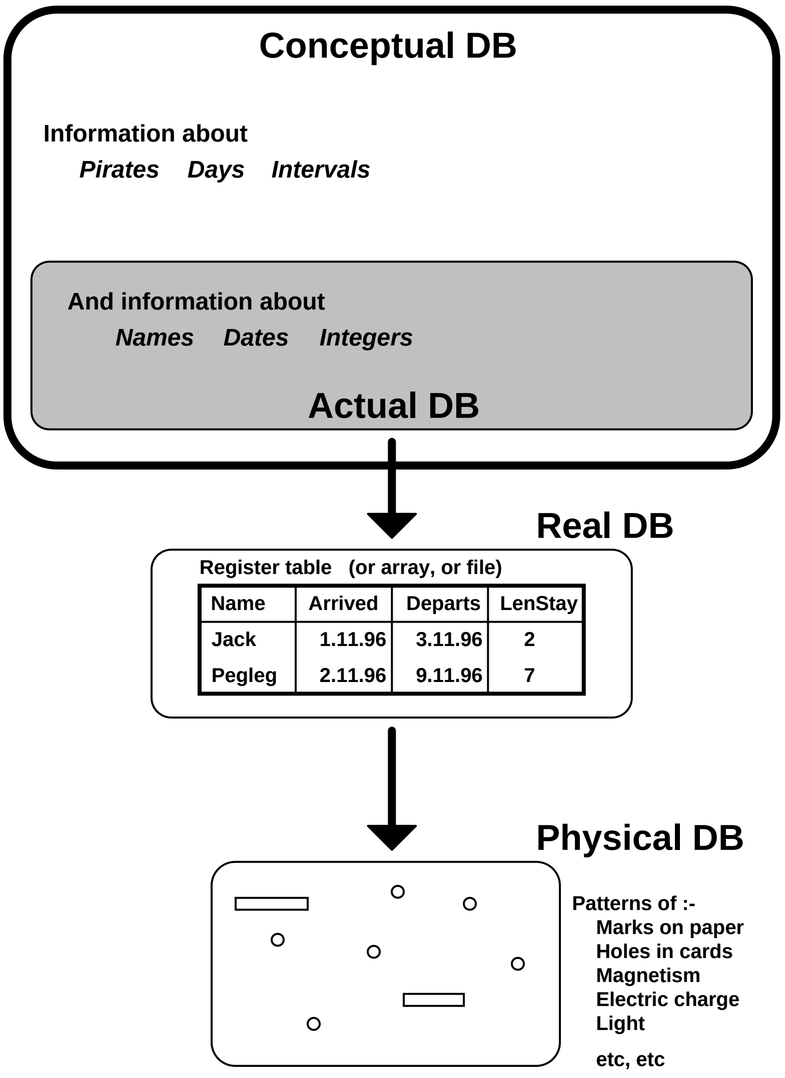
Point 14
Any database has at least two more descriptions :

a) as a **real database**,
described as tables, files, records, etc.;

b) as a **physical database**,
described as patterns of magnetism, charge, holes, marks, etc.

The real and physical database are transformations of the actual database. See Figure 3.1.2.2 below.

Figure 3.1.2.2 Four descriptions of the Plunder Inn register



Name	Arrived	Departs	LenStay
Jack	1.11.96	3.11.96	2
Pegleg	2.11.96	9.11.96	7

It is now time to consider the structure of the information held in conceptual databases. Suppose we sample the Plunder Inn register, or more accurately, an instance of the register, and find that the owner is being reminded of the following items of information.

Example 19.2

```
'Jack arrived on 1.11.96'
'Jack departs on 3.11.96'
'Jack is staying for 2 days'

'Pegleg arrived on 2.11.96'
'Pegleg departs on 9.11.96'
'Pegleg is staying for 7 days'

'Morgan arrived on 7.11.96'
'Morgan departs on 17.11.96'
'Morgan is staying for 10 days'
```

The items have been written out in the usual human language style. For the moment, it does not matter whether these information items come from the actual database or not. If they do then "Jack", "Pegleg", etc are character strings. If not, then "Jack" is a convenient symbol used here to mean a certain pirate, distinct from certain other pirates, etc. (c.f Example 19.1)

One feature of this sample is that some items have something in common. For instance, there are three items concerning arrivals that have "arrived on" in common but talk of different pirates and days (or names and dates). For items that have part in common we could replace the parts that differ by convenient symbols to give a **generic item**. One way to do this for our sample gives three generic items :

Example 19.3
'*p* arrived on *a*'
'*q* departs on *d*'
'*r* is staying for *n* days'

With these generic items we can say that each item of information in the sample consists of a fixed part, such as "arrived on", and a variable part, such as "Jack" and "1.11.96". Now *p*, *q*, *r*, etc. can be called **variables** as they are the placeholders for variable parts of information items. Variables will always be written in italics to distinguish them from other words. It seems plausible to say that the contents of any database can be described in this way : as a collection of information items each with a fixed part and a variable part. Any awkward cases, such as a personal diary (Ex 11), a museum (Ex 15), an encyclopaedia, and even a handkerchief with a knot in it (Ex 17), can be covered if necessary by the **universal generic item**

'The database owner wishes to be reminded of Ψ'

where Ψ is any information item such as 'Talk to Alan on Friday' or 'There is something I must not forget'.

Point 15
The content of any database can be described as items of information, each consisting of a fixed part, possibly common to many items, and a variable part, peculiar to the individual item.

Note that the generic items given in Example 19.3 are not unique. All the items in our sample could be covered by the universal generic item. Alternatively, there is

Example 19.4
'*p* does *x* on *a*'
'*q* stays for *n* days'

where *x* can be "arrival" or "departure". (The English grammar is not perfect). Unfortunately, there does not appear to be any useful canonical form for describing information by generic items. There are the two extremes, but neither is very useful : one generic item covering all information items, or a separate generic item for each information item.

If sensible values are substituted for the variables in a generic item then the result will be an item of information held in the database. For instance, if "Jack" and "1.11.96" are substituted for *p* and *a* in Example 19.3 then the result is the item 'Jack arrived on 1.11.96' found in our sample, Example 19.2. Clearly, the database need hold only the values "Jack" and "1.11.96" if the users can connect these values to the correct variables in the correct generic item. The users can then reconstruct the information item 'Jack arrived on 1.11.96'. Of course, this is the technique used in most database implementations. In the Plunder Inn register the variable parts of the information items in the actual database are stored as instances of the Resident structure. The fixed parts appear as generic items exactly once in the code that implements the reporting facilities. In a telephone directory (Ex 4) the generic items appear once as instructions at the start of the directory; the rest of the directory stores the variable parts in a systematic way.

The generic items need not be held in the database provided they are known to users. For instance, the entry "AB 3 pm" would be quite cryptic in a personal diary (Ex 11) to anyone but the owner. There is no necessity to regard the owner's head to be part of the database. Similarly, the full meaning of the contents of a corporate database (Ex 1) will be known only with the help of the business rules given in the company process manual. There is a component, usually implicit, of any generic item that cannot be held inside the database. This is the component that, in effect, says that this is truly my diary, not yours, that this is truly my company's database, not yours, etc.

Point 16
The fixed and variable parts of the information items in any database can be separated, if desired. The variable parts must be held in the database. The fixed parts need not be (though they should be held somewhere).

If the fixed and variable parts of information items are to be separated then it is vital that the information can be reconstructed on demand and without error. The values and generic items must be linked together in a way that ensures that this can be done. In practice, much of the necessary linking is done by the recording medium.

Example 20
For instance, consider a sheet of paper holding exam marks. It has two columns with headings at the top :

Name	Mark
Jim	53
Carol	73

Ann 64

The heading of the left column is "Name". The heading of the right column is "Mark". Everyone knows that the generic item is

'a student called *name* got the result *mark*'.

Everyone also knows that any two values on the same line belong to the same information item. If "Carol" is to the left of and on the same line as "73" then the information item

'a student called "Carol" got the result "73" '

is held in this database. The reconstruction of the item is made possible by the fact that values can be attached to different positions on the sheet of paper, that these values can be chosen independently, and that the readers know the significance of the positions.

A similar technique is used in computerised databases, including the Plunder Inn register, though typically with more indirection. Incidentally, the technique is not confined to databases. An expression such as 7 ÷ 4 is interpreted correctly because of conventions about the relative position of symbols. If this page makes sense then conventions about the positions of words on the page have played a vital part.

The rest of the necessary linking is provided by rules that allow information items to be deduced from other information items. In particular, this is the only way that items in the actual database can be linked to items in the rest of the conceptual database. One kind of rule allows us to link character strings such as "Jack" to the objects they refer to such as a particular pirate. First we must make a distinction that was deliberately ignored in Example 19.2. The information item 'A certain pirate arrived on a certain day' concerns pirates and days and is not in the actual database; the companion item ' "Jack" arrived on "1.11.96" ' concerns names and dates and is in the actual database. This second item is phrased badly. Character strings do not arrive, nor depart. It would be more accurate to write ' "Jack" then "1.11.96" says something about an arrival', but this would be tedious. As an expedient we will sometimes use the prefix "act-" to indicate an item from the actual database, as in ' "Jack" act-arrived on "1.11.96" '. The two information items are described by different generic items so the three generic items of Example 19.3 should really be six :

Example 19.5
```
'p arrived on a'            'pv act-arrived on av'
'q departs on d'            'qv act-departs on dv'
'r is staying for n days'   'rv act-is staying for nv days'
```

If the owner of Plunder Inn is to be reminded that Jack arrived on 1.11.96 on seeing ' "Jack" act-arrived on "1.11.96" ' then there must be a rule linking "Jack" with a pirate and "1.11.96" with a day. One way to express this rule is to say that the conceptual database contains these two items of information :
```
'pirate Jack has the name "Jack" '
'day 1.11.96 has the date "1.11.96" '
```

There are therefore three more generic items to declare :

Example 19.6
```
'pirate s has the name sv'
'day t has the date tv'
'interval u has the value uv'
```

The rule says that if an instance of

'*pv* act-arrived on *av*'

is in the actual database then instances of

'pirate *s* has the name *sv*',	where the same value is substituted for *pv* and *sv*,
'day *t* has the date *tv*',	where the same value is substituted for *av* and *tv*,
'*p* arrived on *a*',	where the same value is substituted for *s* and *p*,
	and the same value is substituted for *t* and *a*,

are also in the conceptual database. There are similar rules for departures and intervals.

These rules are an example of an identification scheme. Each pirate, day, and interval of interest has an identifier, alias label, which is used in the actual database. In this case the scheme is a simple one. A more complicated case occurs in the identification of roads and houses. A road might have the name "Green Lane"; a house in that lane might have the address "3, Green Lane". The house's identifier has two components : a number and the name of the road. The simplest case occurs when no identifier is needed at all. A dictionary of acronyms might contain the acronym "UMIST", which needs no identifier or, equivalently, is its own identifier. A Greek vase in a museum showcase (Ex 15) also represents itself if it is a special vase; alternatively, it might be the identifier for a large class of vases, including itself.

So far, nothing has been said about uniqueness. If ' "Jack" act-arrived on "1.11.96" ' is recorded in the actual database then by the above rule there is some pirate named "Jack", but there could be many named "Jack". Another kind of rule asserts some sort of uniqueness between pirates and their names so that the owner of Plunder Inn knows who to charge for each visit. In fact, the implementation of the Plunder Inn register (Ex 19) ensures that the register never has any two entries mentioning the same name. If a pirate wishes to visit Plunder Inn again he must wait until the record of the previous visit has been deleted, presumably after it has been paid for. Note though that the implementation could be derived from any one of several different rules :

a) no two pirates will ever have the same name (like Chinese emperors),

b) no two pirates alive at the same time will have the same name (like British race horses),

c) no two pirates with the same name will ever stay at Plunder Inn, or

d) the management ensures that no two pirates with the same name stay at Plunder Inn at about the same time.

Note also that the uniqueness of identification is not always possible and not always desirable. A railway ticket machine might record that someone bought a certain ticket but without being able to say who. A record of children's heights might say that some child is 150 cm tall but not record the child's name for ethical reasons.

To sum up :

Point 17
Rules must exist so that information items in the database can be reconstructed. In particular, rules are needed to

a) Reconstruct information items if their fixed and variable parts are separated;

b) Construct items in the conceptual database given the content of the actual database.

The "Plunder Inn" computer program could be used to hold other kinds of information, without change. For example :

Example 19.7
Record holiday visits, such as
'Visited Pegleg for 7 days from 2.11.96 to 9.11.96'

Example 19.8
Record activities in a work plan, such as
'Revise Maths for 7 days from 2.11.96 to 9.11.96'

Example 19.9
Record observations of unicorns, such as
'Unicorn E23 observed for 7 days from 2.11.96 to 9.11.96'

As unicorns do not exist the database in Example 19.9 will remain empty. Nevertheless, the database exists and is capable of recording many different unicorn observations.

Point 18
The variable part of the actual database can be described and studied on its own as an object common to many conceptual databases. A textbook on database implementation could do this for instance. The real database and the physical database can also be studied independently, of course.

This point leads immediately to

Point 19
The fixed parts of the information items play a vital role when interpreting the contents of a database. So too do the rules connecting the actual database to the rest of the conceptual database.

3.1.3 A model of database instances

In this section we will construct a set-theoretical model of any instance of any conceptual database. The model is to serve two purposes. The first purpose is to demonstrate the key properties that any object purporting to be a database must have. The contents of a database can be organised in various ways. For instance, a record of students' exam marks over several years could use a separate sheet of paper for each year, or a separate sheet of paper for each student, or be held in a single table for all years and all students. The model should be a simple one that ignores such implementation concerns as far as possible. The second purpose of the model is to act as an adjunct to any model of conceptual data models. If a data model says something about a database then presumably it says something about the database's instances, which should therefore be modelled. Simplicity may be helpful for this purpose as well.

The model is based on the use of generic items and the linking of values to the variables in these generic items. As shown in earlier sections, any database instance can be described this way, though perhaps not very informatively in some cases (Point 15). A warning is necessary here. Generic items contain symbols called variables, and so, of course, do the Wffs of set theory. These are different kinds of variable, belonging to different languages. The variables in generic items are modelled here as sets, which may be mentioned in Wffs. When this is done they are constants in the language of set theory and will be written in non-italics to remind us of this.

Just as information items are described as having a fixed part and a variable part, so, too, will a database instance be described as having a fixed part and a variable part. The fixed part consists of generic items, rules enabling the contents of the conceptual database to be derived from the contents of the actual database, and any additional rules imposed by the database owner. Typically, the fixed part will be common to many database instances. The variable part consists of values to be substituted in generic items when reconstituting information items. We will start with the variable part.
As noted in Point 17, it is vital that any values stored in the recording medium can be linked back to the correct variable in the correct generic item. As variables are simply placeholders they can be chosen so that no two generic items associated with the instance use the same variable. The model will assume this has been done. Thus each value need only be linked to the correct variable. This linkage is modelled by a couple. For instance, the link from the value "Jack" to the variable *pv* is modelled by the couple ⟨pv, "Jack"⟩. The recording medium has been abstracted away so no distinction needs to be made between the actual database and the rest of the conceptual database.

It is equally vital that it is known which values stored in the recording medium belong to which information item. Values must be grouped together in some way. The model combines grouping and linking by modelling the variable part of each information item as a set of variable, value couples. For instance, one such set is

$\mathrm{k} =_{d} \{ \langle \mathrm{pv}, \text{"Jack"} \rangle, \langle \mathrm{av}, \text{"1.11.96"} \rangle \}$.

Given k and the generic item '*pv* act-arrived on *av*' we can reconstruct the information item ' "Jack" act-arrived on "1.11.96" '. This item belongs to the actual database, but once again the recording medium has been abstracted away so the model applies to the whole conceptual database.

The variable part of any information item can be modelled as a set of couples in this way. When this is done, each variable appearing in the associated generic item is linked to exactly one value. Thus the set of couples can be described as a family whose index set is the set of variables appearing in the generic item. E.g The set k above can be described as the family $(\mathrm{k}_i \mid i : \{\mathrm{pv}, \mathrm{av}\})$. But a family is one of the standard ways of representing a tuple. To no-one's surprise, the variable part of each information item has been modelled as a tuple. E.g The set k can be described as the tuple (pv $\mapsto$ "Jack", av $\mapsto$ "1.11.96"). The variables appearing in generic items can now be called indexes when convenient.

With this representation of tuples there is nothing special about unary tuples, alias 1-tuples. The set { ⟨c, Jack⟩ } models the variable part of an information item such as 'Jack is a regular customer', whose generic item, '*c* is a regular customer', happens to have only one variable. Note that modelling it as the singleton set { Jack } would lose the link back to the right generic item. However, there is a special case. The set { } is a set of couples and it represents the nullary tuple, alias 0-tuple. It models the variable part of an information item whose generic item has no variables. Unfortunately, the 0-tuple, being empty, has no means to indicate which generic item it is to be associated with. Luckily, a generic item with no variables such as 'There is a regular customer' is properly an information item and will be treated as such. There is no reason to attribute a meaning to the presence of the 0-tuple in the model of a database instance, and no reason to allow its presence.

Finally, it is vital that it is known which tuples encoded in the recording medium belong to the database instance. (There may be some that do not belong : think of a piece of paper with something crossed out). The variable part of each conceptual database instance is modelled as a set of tuples, but to see that this is a satisfactory model we must check three pre-requisites. First, the recording medium has been abstracted away so no distinction needs be made between actual and conceptual database. Second, tuples modelling distinct information items differ by values or by index set, by decree, and so are distinct themselves. Third, suppose a tuple occurs twice in the recording medium. What does this signify? One possibility is that tuples are replicated for security. The model can ignore replication as it is a feature of the physical database. The other possibility is that the multiple occurrences are meaningful. But then the generic item has been written incorrectly. It should really be 'case *n* of …', as when "Jack" is written as "Jack I", "Jack II", etc. Thus multiple occurrences are not significant and should not be modelled.

Point 20
The variable part of any conceptual database instance can be modelled as a set of tuples, where each tuple is represented as a family with a carefully chosen index set. The nullary tuple is excluded.

This is the model we will adopt for this work. There are other models that would do the job, of course. Some of them are outlined in the next sub-section.

We will now turn to the fixed part of a database instance : generic items and rules. We will see in Section 3.2 that the job of a conceptual data model is to specify generic items and rules so we will not attempt to model them here. We will assume here that any model of conceptual data models can be used to model the fixed part of a database instance.

However, we will note a consequence of having rules. Presumably the rules can be translated into a form that enables us to apply them in our model of the variable part of database instances. We will assume that there is an implicit rule requiring each tuple to have a corresponding generic item in the fixed part. Thus for *any* set of tuples we can determine whether the translated and implied rules are obeyed or not. Instances will be classified accordingly. We will say that a set of tuples models a **legitimate** conceptual database instance if all the translated and implied rules are obeyed; it models an **illegitimate** instance otherwise.

Notice that the model of instances lets us describe a transition from one legitimate instance to another via a sequence of transient illegitimate instances, if that is convenient. It also lets us talk of tuples that are not present in any legitimate instance.

3.1.4 Some alternative models

The previous section modelled the variable part of a database instance as a set of tuples. Other organisations are possible, of course. We will describe two that are well-known, and show why we believe they are inappropriate for our purposes.

Tuples sharing the same index set can be collected together and indexed to give a family of sets of tuples. Obviously, this organisation is better described as a family of relations, or of relation graphs if domains are defined. We now have two layers of indexing. A rule concerning the values assigned to a certain tuple index, say pv, must also quote the index, say α, of the set containing the relevant tuples. In effect, the tuple index is now referenced by the compound value α.pv.
A further change to the organisation is to dispense with tuple indexes altogether by using nested couples to hold nothing but values. E.g $\langle\langle a, b\rangle, c\rangle$ would be a ternary tuple. However, there is then a family of projection functions associated with each relation, such as $\pi_{\alpha,\, pv}$ and $\pi_{\alpha,\, av}$ to link couples such as ⟨"Jack", "1.11.96"⟩ to the generic item variables *pv* and *av*. Now projection functions must also be quoted in rules. In addition, a definition of 1-tuples is needed and also a definition of the 0-tuple if it is to be mentioned.

Extra index layers and projection functions are unlikely to make translated rules easier to read. They are not essential and they might hinder the recognition of objects that can be usefully described as databases. In short, they are unnecessary implementation details.

Point 21
Any database instance can also be modelled as a family of relations, but at the expense of introducing additional layers of indexing and additional terms in translated rules.

A different organisation is seen in the usual implementation of "relational" databases. The (real) database is a family of tables. Each row in a table has the appearance of a tuple; column headings can be thought of as indexes. However, some tables allow certain columns to hold Blank, alias Null, values. One way to model the presence of Blanks is to say that a row is a partial function from the index set to values. A Blank indicates an index that is not in the definition domain of the function. Another way is to say that a row is a total function and that Blank is a special value that cannot be confused with any other value.

However, it seems simpler to say that Blanks are an artefact of an encoding method that allows a row to hold more than one tuple. A Blank indicates that a tuple is absent in the row. For instance, suppose we have a simple table of sines and tangents. Two rows of the table might be :

r1 $=_d$ [45, 0.707, 1.000]
r2 $=_d$ [90, 1.000, Blank]

The information items belonging to the actual database encoded in r1 are

' sin 45° is 0.707 ' and ' tan 45° is 1.000 ',

while in r2 there is only

' sin 90° is 1.000 ';

no tuple occupies the rightmost position in the row as 90° is not in the definition domain of tan.

There would be no benefit incorporating such complications into a model of conceptual database instances.

Point 22
Tables in relational databases can hold Blank, alias Null, values. Blanks are deemed here to be an artefact of the encoding method used in the real database.

3.1.5 Credits

There are four ideas in NIAM that have been adapted for use here (Nijssen & Halpin [1989]; 2nd edition : Halpin [1995]).

The first idea is the declaration that the contents of a database can talk of people and days as well as of names and dates. This has lead us to make a distinction between the conceptual database and the actual database. The word "conceptual" is taken from the NIAM term "conceptual data model", of course.

The second idea is the declaration that the contents of a database are "facts" such as 'Jack arrived on 1.11.96'. We have used the term "information item" instead of "fact" to avoid confusion when talking of "facts" that are untrue or not recorded in the database.

The third idea is NIAM's use of anonymous placeholders, as in '… arrived on …', to describe classes of information items. We have given names to the placeholders, called them variables, and used them as indexes. We also call the resulting description a "generic item".

The fourth idea is that Blank values are an artefact of relational database implementation. The final step in the NIAM development process is to execute an algorithm that translates the conceptual data model into a relational database schema. The algorithm defines tables that can have Blank values. This is entirely natural and is clearly the consequence of representing several partial functions in one table.

Codd modelled databases as collections of evolving relations (Codd [1970]). Note, though, that he modelled the actual database, not the conceptual database. His objective was to decouple application software from the details of the real database.

3.2 What is a conceptual data model?

If we wish to design a new product we need to know which of its characteristics should be fixed in advance. We use examples to show that some databases need certain characteristics fixed in advance, while others have no fixed characteristics that would justify the use of a database design technique (Section 3.2.1).

If we wish to discover reasons for choosing the NIAM database design technique we need to know how NIAM purports to fix the characteristics of a database. We use a continuing example to illustrate this (Section 3.2.2). However, there are awkward cases to be investigated. Some are improper NIAM constructions; some are cases for which NIAM is not suited (Section 3.2.3).

We then list some preliminary conclusions (Section 3.2.4). Finally, we state what is required of any model of NIAM conceptual data models (but do not introduce a model as such) (Section 3.2.5).

As in the previous section, many of the key ideas have been adapted from the work of other authors : credits are given in Section 3.2.6. Some terms are used here and later on with special or revised meanings. They are highlighted in bold at the point where they are introduced.

3.2.1 Some examples

Here are some examples of databases, some of them replicated from Section 3.1. What might it be useful to prescribe in advance before creating and using each of these databases?

Example 1
A handkerchief with a knot tied in it, reminding the owner that something is to be remembered.

On the one hand, this is an ad hoc database created when needed, with no prior description at all.

On the other hand, this is a database that has exactly two legitimate instances : knot or no knot. All instances and their meanings have been described so it has already been designed.

Example 2
A kitchen notice board containing a list of things to be done, who is to do it, exhortations, quotations, general remarks about life, etc. For instance :

```
Jack departs on 3.11.96
King Harold was killed in 1066
E = m c²
Vote early, vote often!
```

The contents of this database are completely free. It is difficult to imagine anything except the recording medium that could be prescribed in advance. Notice that the information in this database instance is incomplete. For instance, the reader is assumed to know that Harold was an English king, and that 1066 is a year number, AD.

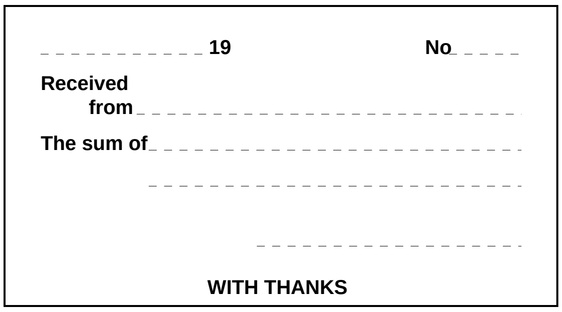
Example 3
A shopping list, containing reminders of things to be bought, showrooms to be visited, etc.

The owner of this kind of database will regard the contents as completely free and not prescribed in advance. Even if the information is usually to do with purchases it can still be quite diverse. It would be difficult to describe all cases in advance. For instance, any of these is possible : 'Buy potatoes', 'Buy bone for Fido', 'Buy present for Ann costing under £10'.

Point 1
By their nature or by their mode of use, some databases have nothing prescribed in advance.

However, this is certainly not true of all databases.

Example 4
A cash receipt book bought from a newsagent. Alternate sheets are pre-printed thus :


_ _ _ _ _ _ _ _ _ _ _ 19 No_ _ _ _

Received
from_ _

The sum of_ _

_ _

_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

WITH THANKS

The pre-printed pages display the fixed part of this database; the dotted lines are placeholders for values which may vary from one receipt to another. When values are written on the dotted lines then the result is an item of information that has been inserted into the database. For business and legal reasons no other kind of information is allowed into this database. Moreover, the database users are required to write sensible values on the dotted lines. For instance, they are not allowed to issue a receipt saying "Received from £25.97 The sum of Donald Duck".

There are more rules. For instance, the database owner requires that the "Received from" name is that of someone who has paid and that the signature is of someone who is authorised to issue receipts. Thus, here is an example of a database that has many things prescribed in advance : the fixed part of any information item, the values allowed in the various placeholders, the business rules governing changes to the database. Some of this may be implicit but will still be clearly understood.

This is also an example of a database that is duplicated for security. For each receipt written on a pre-printed page there is a carbon copy on the adjacent counterfoil. There is no printing on the counterfoil pages so the values written on them must be linked to the placeholders in the top copy solely by their position on the page.

Example 5
A telephone directory, containing names, addresses, and telephone numbers.

Example 6
A dictionary, containing words and their meanings.

In both of these examples the kind of information permitted in the database is fixed in advance. In fact, if any other kind of information were present then it would not be a telephone directory or a dictionary. The values that can be seen by users are also restricted. Note, though, that telephone "numbers" are character strings whose characters match the symbols printed on telephone buttons. These could change in the future, so changing the permitted values.

Example 7
A corporate database, containing details of a company's purchases, sales, stocks, employees, salaries, etc, etc.

The owner of this database uses it to control the business. It must hold certain information else the company will disintegrate. Equally, it must not hold any other information else money is being wasted.

The owner requires the information in the database to be relevant to the workings of the business. The workings are described in the company's process manual or are known to key staff (if any are left after downsizing and outsourcing). The information immediately visible in the database must be clearly linked to the people, goods, invoices, etc described in the process manual.

The owner wishes to reduce the likelihood of human errors by the users. Although errors cannot be prevented entirely, he wishes any errors to be simple ones such as a wrong spelling or a wrong number rather than an illogical tangle that is difficult to correct when finally noticed.

Thus, here is an example of a database that must be thoroughly prescribed in advance. Furthermore, the connection between the database and the operations of the company must be well known and well maintained for the lifetime of the database.

Point 2
Some databases are prescribed in advance. Any of the following could be prescribed :

- Rules determining legitimate database instances, including :
  - permitted kinds of information item,
  - permitted values,
  - permitted combinations of information items;
- Rules determining permitted transitions between instances;
- The links between the contents of the database and
  - the company's business objects and processes.

It is quite likely that the description of a corporate database will cover several thousand different kinds of information.

Point 3
One reason to prescribe a database in advance is that the mass of detail is overwhelming. This is also a reason for desiring automated assistance.

It is also quite likely that resources must be procured before the database can be used : application software, pre-printed receipt books, information-gathering machinery or organisation, even a kitchen notice board.

Point 4
One reason to prescribe a database in advance is that dedicated resources must be procured before the database can be used. Something must be known about the database in order to do this properly.

3.2.2 Case study : the Plunder Inn register

We will continue the study of the Plunder Inn register of residents that we started in Section 3.1. How would a NIAM-style conceptual data model prescribe this database? Does the data model have a well-defined meaning in this case? Are there any unexpected assumptions?

Example 8
To repeat, this example is taken from a C++ programming exercise and the essentials of the program's specification are :

"*Plunder Inn is a shelter for pirates. It maintains a register recording for each resident his/her name, date of arrival and date of departure (only one stay at most per pirate is recorded). You are … to represent this register as an array of records.*" "*… the program will prompt the user to input a name, a date of arrival and a length of stay.*" "*The length of stay will be a nonnegative integer number of days.*"

The format of individual records in the register is specified :

```
struct Resident
  { NAME name;
    DATE arrival;
    int  LenStay;
    DATE departure;
      // Invariant: arrival plus LenStay = departure
  };
```

(The types NAME and DATE are also specified).

We will continue to pretend that the requirements of the Plunder Inn database are real. One way to prescribe it in advance is to give a general description followed by detailed record definitions as is done above. This is sufficient to enable the owner of Plunder Inn to interpret the contents of the database and to instruct users in its operation. It is also sufficient to enable the program to be designed and written. However, this begs the question of where the record definitions come from. It also restricts the implementation. There may be other ways to do the job that would be equally satisfactory, e.g a relational database; a registration book; a blackboard.

Another way to prescribe it in advance is to say that the conceptual database must be able to hold certain information items, and no others. Typically, there will also be rules saying what combinations of information items are permitted in database instances. Now the encoding of information in the database, for instance as records or tables, can be treated as a separate concern to be decided later. Perhaps the database will be implemented more than once, in different ways.

This is the technique used in NIAM. The first step in the NIAM process is to write example information items and then to classify them. The result is a set of what we have called generic items. Recall that a generic item holds the fixed part common to many information items, with placeholders, alias variables, to show where these information items differ from each other. Recall also that the conceptual database holds information about pirates and days as well as about character strings and numbers. The information immediately visible to users belongs to the actual database; the rest requires an act of imagination.

On doing this for the Plunder Inn register we get the generic items listed in Example 8.1. As before, we use the prefix "act-", for actual, where needed to distinguish information items that belong to the actual database from those that do not.

Example 8.1
Permitted generic items :

```
'p arrived on a'
'q departs on d'
'r is staying for n days'

'pv act-arrived on av'
'qv act-departs on dv'
'rv act-is staying for nv days'

'pirate s has the name sv'
'day t has the date tv'
'interval u has the value uv'
```

There is an obvious question here : why these particular generic items? There is a design technique that could produce this list; it will be described later on.

Even if an information item is described by one of these generic items it can still be inappropriate. For instance, we do not want the Plunder Inn register to say that a certain pirate arrived on a certain mountain. First, the owner is unlikely to know what this means nor want to be reminded of it. Second, it is unlikely that one mountain can be distinguished from another by a date. The next step in the NIAM process is to say which values can be substituted for which variables in the generic items, as follows :

Example 8.2
Permitted values :

```
p, q, r, s       : only pirates
a, d, t          : only days
n, u             : only intervals

pv, qv, rv, sv   : only names
av, dv, tv       : only dates
nv, uv           : only integers
```

It is clear what days and intervals are. For names, dates, and integers there is a choice. Names could use the English alphabet, the Greek alphabet, etc, and might even include arbitrary symbols such as trade marks or logos. Dates could be day/month/year values, Julian day numbers, etc. Integers could be binary, decimal, Latin, etc. It would be wise to leave the choice until more is known about the capabilities of the equipment implementing the database.

> **Point 5**
> A database may be prescribed in advance in a way that leaves some choices open, with the final choice to be treated as a separate implementation concern.

Note that we are no longer prescribing a database. We are prescribing a class of suitable databases, each with its own class of permitted instances.

It is not at all clear what a pirate is. Is there a way of proving that someone is or is not a pirate? One answer is that if someone is staying at Plunder Inn then he, she, or it is automatically a pirate. The status of anyone else does not matter. It is the Plunder Inn receptionist who classifies objects as pirate or not-pirate, but only when a decision is needed. Thus the statement that only pirates can be substituted for the variables *p*, *q*, *r*, *s* is really a statement that only objects admitted by a certain decision procedure can be substituted. That decision procedure is implemented outside the database proper. Although the procedure might be prescribed in advance it can include the exercise of human judgement and an element of randomness. That said, it is simpler to say "only pirates". No harm is done provided the database owner is aware of the need to operate a suitable decision procedure.

It is a NIAM principle that each variable of each generic item is restricted to a fixed class of objects. In some cases it is clearly possible to specify a well-defined fixed class of objects, even if the final choice of class is left until later. For instance, all integers, all days, all names. In other cases it is not so clear. Is there a class of all pirates, fixed in advance? The view taken here, and implied in the NIAM literature, is that it does not matter. To say that there is a fixed class of all pirates is an approximation or idealisation that is reasonable. Provided the class is large enough it will do to describe all practical use of the database. The class models all the objects that could conceivably be admitted by the Plunder Inn receptionist.

> **Point 6**
> NIAM restricts each variable of each generic item to a fixed class of objects. In some cases this is an approximation or idealisation, but one that may be reasonable. If it is not reasonable then NIAM should not be used.

A NIAM Fact Type is the class of all those information items described by a particular generic item that have permitted values. We will also use the term **Fact Type** when the information items are represented by tuples. As domains are fixed then any such Fact Type is a cartesian product.

The statement of the permitted generic items given in Example 8.1 and of the permitted values given in Example 8.2 determines a class of information items. Every information item in any legitimate instance of the Plunder Inn database belongs to this class. Thus Example 8.1 and 8.2 define outer limits within which the database may evolve. Of course, an instance can be within these limits and still be illegitimate; these could be said to be the more interesting illegitimate instances. Any additional rules that legitimate instances must obey can assume that the instance is within these limits, if this is convenient.

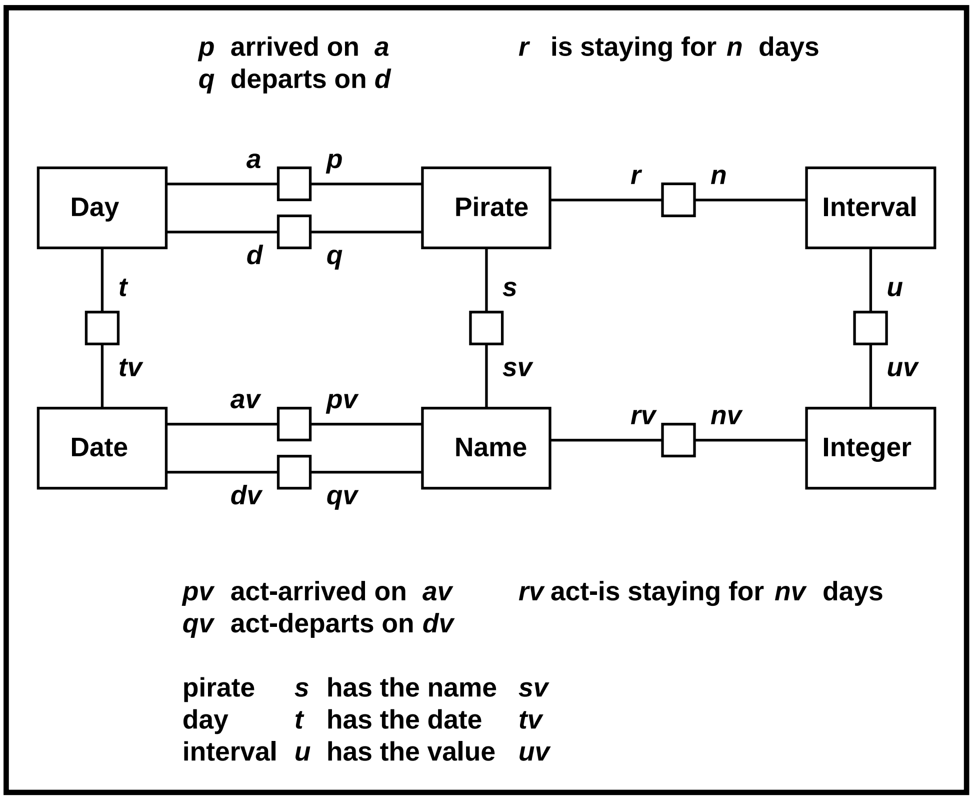
Point 7
One way to prescribe a database in advance is to prescribe a class of information items to act as outer limits within which the database must evolve. This is the technique used in NIAM. Database evolution will usually be subject to additional restrictions.

The generic items and value restrictions given in Example 8.1 and 8.2 are explicit and clearly stated, but would the Plunder Inn owner and receptionists find them easy to check? Are there any typing errors? Are there any generic items obviously missing? Typically, people find it easier to assess a diagram so NIAM uses a pictorial notation.

The next step in the NIAM process is to transform the design information given in Example 8.1 and 8.2 into a diagram. However, there are some implicit conventions used in NIAM diagrams that need to be elucidated. Rather that create the diagram immediately we will do the transformation in four steps, ending with a conventional conceptual data model in a particular NIAM dialect.

In the pictorial notation used here there is a distinct small square for each generic item; there is a distinct large rectangle, suitably annotated, for each class of permitted objects, alias permitted values; and there is a distinct line for each variable, joining the symbol representing its generic item to the symbol representing the class it is restricted to. The first picture, in Figure 3.2.2.1, is unconventional. Each line is annotated with its variable, and the text version of its generic item is written nearby.

Figure 3.2.2.1 **The permitted classes and generic items (an unconventional picture)**


p arrived on *a*
q departs on *d*
r is staying for *n* days
Day
Pirate
Interval
Date
Name
Integer
a
p
d
q
r
n
t
tv
s
sv
u
uv
av
pv
dv
qv
rv
nv
pv act-arrived on *av*
qv act-departs on *dv*
rv act-is staying for *nv* days
pirate *s* has the name *sv*
day *t* has the date *tv*
interval *u* has the value *uv*

Notice that Figure 3.2.2.1 makes it clear that the thing that a pirate arrives on (a day) is certainly the same kind of thing that he departs on, which is reasonable, and is probably not a thing that can name him, which is also reasonable. Furthermore, if "Pirate" is mistakenly written as "Prate" then readers might wonder what "Prates" are but are still in no doubt that they participate in four kinds of information item. Typing errors do not change the structure of the picture. This is a useful error management feature.

Notice also that Figure 3.2.2.1 reveals two hidden assumptions in Example 8.1 and 8.2. The first assumption is that the same text has not been used in Example 8.1 to describe two distinct generic items. For the pictorial notation we declare that there is a distinct generic item for each small square. (More accurately, for each distinct position on the paper holding a small square). The second assumption is that each variable quoted in Example 8.2 belongs to exactly one of the generic items of Example 8.1. For the pictorial notation we declare that there is a distinct variable for each line.

Point 8
A pictorial notation allows two concerns to be separated. Symbols can be used to assert the existence of distinct objects, while text annotation can be used to communicate the meaning or use of those objects.

It is a common convention to use singular words, such as Day and Pirate, for classes of permitted values rather than plurals such as Days and Pirates. This allows us to explain a generic item by saying "a Pirate arrived on a Day" instead of the more cumbersome "one of the Pirates arrived on one of the Days". It has no other significance.

The next step in this sequence of pictures attends to three matters. First, the variables will not mean much to lay people and the generic items clutter up the picture. They are replaced by annotation that conveys to lay people what the intention is, in a way that makes it obvious to experts what they should do to reach Figure 3.2.2.1 or the equivalent text in Examples 8.1 and 8.2.

Second, we must say which information items are held in the actual database. Remember that these items are immediately visible to users. They are the only items that need to be translated by the implementers into the contents of the real database and the physical database. Here they are the information items whose generic items include the prefix "act-" in Example 8.1.

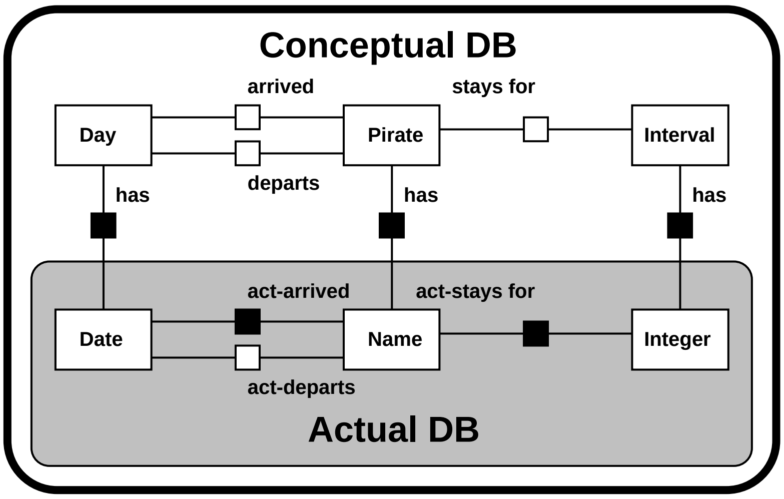
Note that this is a design choice, though there are obvious practical considerations. For instance, we could decide to remember which pirate has which name by sticking a label on each pirate and deeming this to be part of the actual database. (Remember, we are not committed to using a computer yet). However, we would then not be able to copy the database for archiving.

Third, we can start to show how the presence of some information items in the conceptual database determines the presence of others. Remember that each small square in Figure 3.2.2.1 stands for a distinct generic item. In any database instance there will be a set, possibly empty, of information items described by this generic item. In some cases the set is uniquely determined in all legitimate database instances by information items of other generic items. For instance, if a pirate arrives on 1.11.96 to stay for 2 days he can only be leaving on 3.11.96. NIAM usage is to say that the corresponding Fact Type is a derived Fact Type, but note that it is parts of database instances that are derived, not the Fact Type itself. A square is coloured white to indicate that the corresponding set of information items is uniquely determined by others. (In all legitimate instances, that is; anything is possible in an illegitimate instance.) A square is coloured black when this is not so, though the legitimate sets of information items may be restricted.

Of course, saying that something is uniquely determined is not enough. We need to say what rule is used to determine it. This will be discussed later on but even now the white squares remind us of some rules that will be needed. We can also see that information about departures need not be stored in the real database. It is sufficient to reconstruct the information on demand : here is another choice to be treated as an implementation concern.

The result of doing these three changes is shown in Figure 3.2.2.2 below. The text annotation has been simplified. The actual database, containing information items immediately visible to the users, has been highlighted. The white squares indicate the derived information items, those whose presence is determined by other items.

Figure 3.2.2.2 The conceptual and actual databases : database user's view



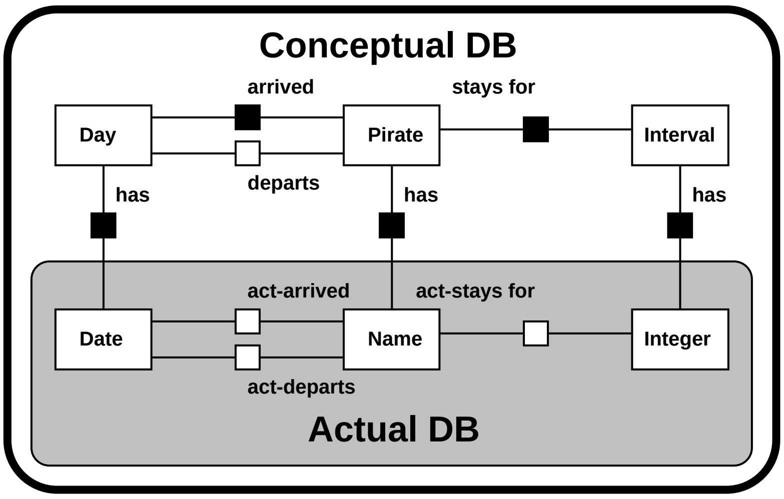
Notice that some of the black squares in Figure 3.2.2.2 are outside the region marked Actual DB. This is typical. The picture, together with some rules about uniqueness and completeness, justifies use of phrases such as "the pirate named Jack", but the Plunder Inn receptionist might still have no way to recognise Jack in all circumstances.

Figure 3.2.2.2 describes the database as seen and understood by its owner and users, though some important information is missing at the moment. The picture shows how the character strings and numbers immediately visible to the users can be used to provide information about pirates and days.

We now return to an earlier question. Why this particular description? Was it so obvious that no other description was considered? This is unlikely, even for so simple a database as the Plunder Inn register. One common design technique is to describe the interesting relationships within the organisation and then deem this description to be a description of the conceptual database. This is the technique used in NIAM.

When this is done for Plunder Inn we now have to say that the arrival of a pirate on a certain day uniquely determines a name and date in the actual database, not the other way round. Some of the small squares in Figure 3.2.2.2 change colour, giving us Figure 3.2.2.3 below.

Figure 3.2.2.3 The conceptual and actual databases : database designer's view



The white squares in Figure 3.2.2.3 that are inside the boundary marked Actual DB describe information items in the actual database. These items enable users to deduce the content of the whole conceptual database, as outlined in Figure 3.2.2.2. The introduction of these white squares into the data model can be delayed and treated as another implementation job to be done later on. Typically, they are not shown in a NIAM diagram. They will be removed, giving us the picture in Figure 3.2.2.4 below. Rather than using a box to surround the symbols describing the actual database we mark the large rectangles with a black triangle and use a convention for small squares. If all the lines from a square lead to marked rectangles then the square represents a generic item of the actual database. As it happens, there are none in Figure 3.2.2.4.

Figure 3.2.2.4 The final conceptual data model, before adding any constraints (Drawn in the "UMIST" dialect of the notation)

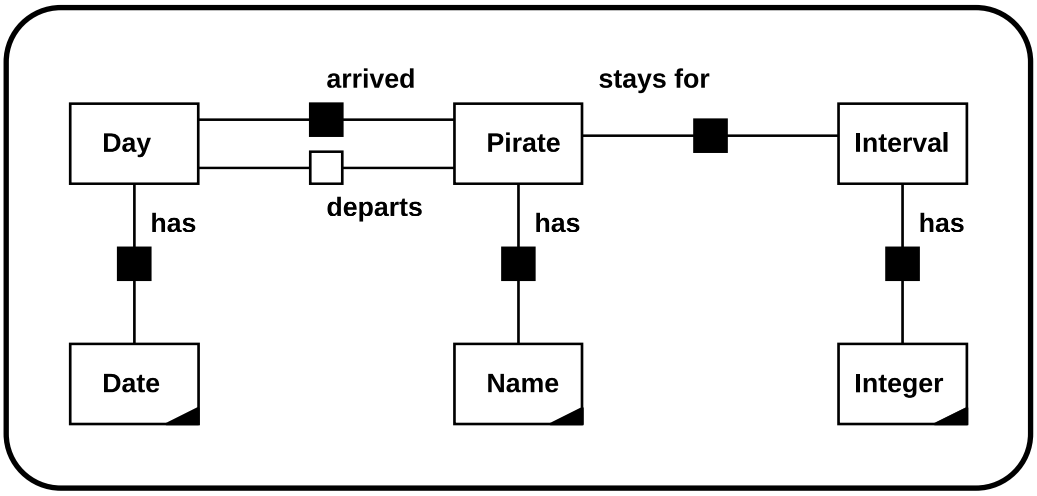


Figure 3.2.2.4 is an example of a conceptual data model drawn in the NIAM style, using the "UMIST" dialect. The process that took us from Example 8.1 and 8.2 through Figure 3.2.2.1 to Figure 3.2.2.4 was essentially a change of notation. Figure 3.2.2.4 without the colouring of squares and marking of rectangles does the same job as Example 8.1 and

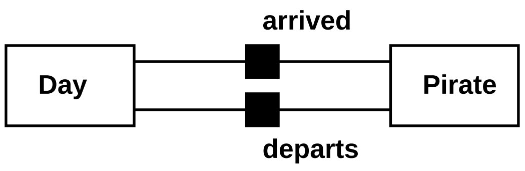
8.2 : it prescribes the permitted generic items and the values their variables are permitted to take. (Subject to deferred choices deemed to be implementation concerns). This part of a conceptual data model will be called the **core** data model.

Point 9
A core data model prescribes the delimiting class of information items. All information items in any legitimate database instance must belong to this class. The prescription is typically subject to deferred implementation choices.

An aside : In theory, the implementers would add the missing generic items to Figure 3.2.2.4, change the square colouring to give Figure 3.2.2.2, then transform the description of the actual database into the description of the real database. In practice, they transform a conceptual data model such as Figure 3.2.2.4 directly into a real database description such as the `struct` declaration given in Example 8. This is unfortunate as the rules for translating from real database back to conceptual database are not as well documented as they might be. Possibly they are not documented at all.

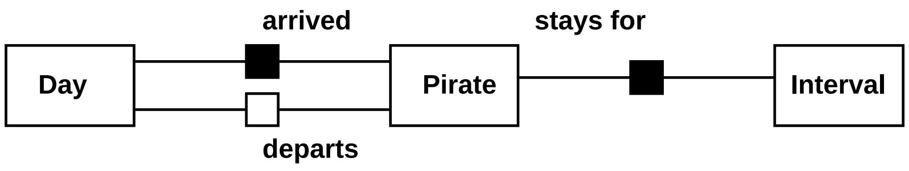
For the last time we ask the question : Where did Figure 3.2.2.4 come from? One plausible answer goes as follows. The owner of Plunder Inn says that he wants a register recording the arrivals and departures of the pirates who visit his Inn. The database designer converts this requirement into the description shown in Figure 3.2.2.5.

Figure 3.2.2.5 The initial conceptual data model



On seeing this the owner remarks that pirates say they are staying for so many days, but that the Inn's staff need to know their departure days. The designer modifies the picture to include a length of stay and showing that the departure date is calculated by the system, giving the picture in Figure 3.2.2.6.

Figure 3.2.2.6 The revised conceptual data model



Values that the database users can write down must be added. The designer chooses the obvious ones : dates, names, and integers. The designer then adds them to the data model, giving the picture we saw in Figure 3.2.2.4.

Point 10
Within the design process a conceptual data model is an evolving object. Typically it will start as the empty model : a blank sheet of paper or a blank screen. Each evolution step is a conceptual data model in its own right.

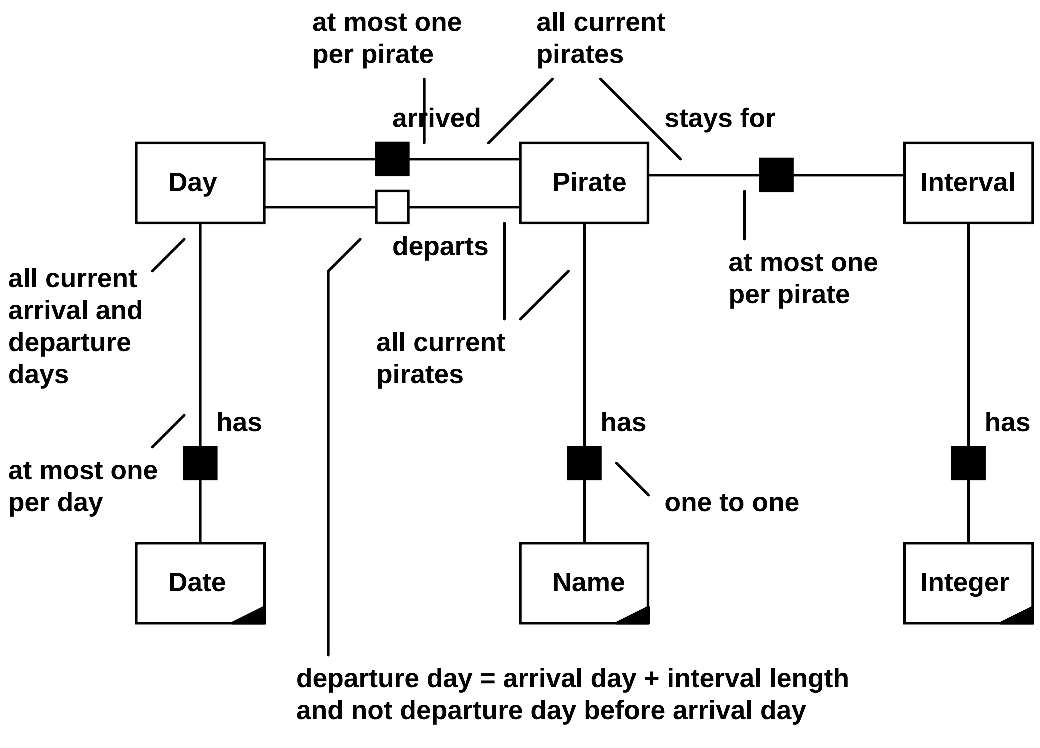
The next step in prescribing the Plunder Inn database is to state the rules, alias **static constraints**, that distinguish legitimate database instances from illegitimate instances. One rule has already been introduced : the core data model restricts information items to a certain class.

Clearly, there must be additional rules. For instance, any pirate staying at Plunder Inn must have a name, at least for registration purposes. It is tempting to say that every pirate has a name. But this might not be true, and it would not matter if it were not true. If a pirate with no name wishes to stay at Plunder Inn then the receptionist will either refuse him admittance or will admit him without recording the visit in the register or will invent a name for this visit. Presumably the owner has a rule saying which the receptionist should do. Whichever is done, it makes no difference to the definition of legitimate database instances.

Point 11
We must distinguish between rules governing the permitted contents of the database and rules governing the users of the database. We declare that a conceptual data model prescribes the former but not the latter.

Some of the additional rules for the Plunder Inn database are shown in Figure 3.2.2.7 below. The rules have been written in an informal and very ad hoc notation. Among other things the picture is saying that if a pirate is mentioned at all in a database instance then his arrival, departure, length of stay, and name must all be recorded in that instance. Also, he cannot be recorded as arriving more than once.

Figure 3.2.2.7 The conceptual data model with some rules added (using an ad hoc notation)



Just as for different information items, different rules can have much in common. Figure 3.2.2.7 has several instances of 'all current …' and 'at most one per …'. For each common kind of rule there is a function which, on being given suitable parameters, will return the formula that expresses the rule. For instance, one function would return the formula

(In a database instance *I*, no two different information items described by

'pirate *s* has the name *sv*'

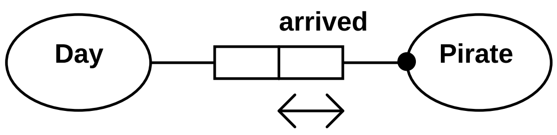
have the same value substituted for *s*)

on being given *s* and the generic item 'pirate *s* has the name *sv*'. The functions are defined for any conceptual data model in a given style. One advantage of using these functions is that a database designer can invoke a precise and usually long statement of a rule with little effort and little likelihood of error. Another advantage is that function application can be included in the pictorial notation. A symbol of a certain kind names the function; its position indicates the parameter(s) to be applied.

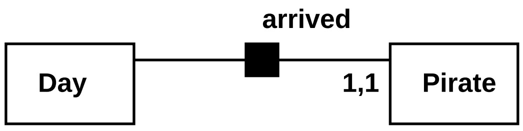
Some examples are shown in Figure 3.2.2.8 below. In part (a) the blob introduces the rule described informally as 'all current pirates have their arrivals recorded'. The double-headed arrow introduces 'at most one arrival per pirate'. In part (b) the conjunction of these two rules is introduced by the "1,1" symbol.

Figure 3.2.2.8 Rules introduced by function application

a) NIAM dialect



b) "UMIST" dialect



Each dialect of the NIAM notation has its own choice of rules that have special symbols. We give a formal definition in Chapter 7 of the two shown in Figure 3.2.2.8(a). This is sufficient to show how any others could be defined.

Point 12
Functions returning common kinds of rule can be defined, and they should be defined. Pictorial notations can include function symbols and function application statements.

Not all rules are common enough to be covered by a standard function. For instance, the rule uniquely determining departure information must be written explicitly. In NIAM such rules are deemed to be part of the data model, even if they are not written in the diagram.

Note that there is no clear distinction between rules needed to make sense of the database and those included to prevent user errors. For instance, the Plunder Inn receptionist might record an arrival but forget to record the length of stay. Does this make the database illogical, or has a business requirement been disobeyed?

There is another kind of rule that is not needed in this example. It can be necessary to declare that certain transitions from one legitimate instance to another are forbidden. For instance, the number of cars that have been owned by a person cannot decrease with time. Such rules are called **dynamic constraints**. Note that privileged users must be able to disobey such rules in order to correct mistakes.

Once all the rules have been written down we have finished the conceptual data model. We can now find a group of implementers, tell them to make suitable implementation choices and then tell them to implement the database. Notice that they can choose to implement the Plunder Inn register on pre-printed paper, as a C++ program, or as a relational database. The conceptual data model is neutral as to recording medium and organisation, while being explicit as to the interpretation of the database contents (given faithful implementation and documentation).

Point 13
Prescribing a database in advance is typically separated into three concerns.
Responsibility for prescribing the features that were listed in Point 2 is :

Conceptual data model :
Permitted information items, legitimate database instances, and legitimate transitions, usually with some implementation choices left open.

Implementation :
Choices left open by the conceptual data model, mostly concerning the description of the actual database.
(Also : The translation of the actual database to and from the real database.)

The organisation's process manual, possibly implicit :
User rules, processes, and procedures.
The links between the conceptual database and business objects and processes

3.2.3 Awkward cases

The pictures used in the previous section to prescribe the Plunder Inn register appear to have a clearly defined meaning. Some, if not all, are obviously incomplete but this is to be expected at intermediate stages in the design of a product. Are there improper pictures?

One kind of impropriety is easily described. Lines that do not have a square at one end and a rectangle at the other end, or equivalent symbols, are clearly improper. We can also ban squares with no lines as these represent dubious generic items with no variables. Clearly, detecting this kind of impropriety is straightforward.

The pictures restrict certain variables to certain classes. What of these classes : can they be improper? Let us investigate some cases that might lead to difficulties. We will start with a simple case.

Example 9
A variable is restricted to the class of all computer programs that are sure to halt for all legal inputs.

It is well known that there is no decision procedure that will select exactly these programs.

Point 14
Some choices for a class of values are simply unwise. For instance, those requiring impractical user rules. There is no obvious general rule that would highlight unwise definitions.

Next, we will consider the extreme case of databases that hold databases.

Example 10
A variable is restricted to database instances.

This is not as unlikely as it might sound :

Example 10.1
Think of an office filing cabinet. One drawer is labelled "Personnel database", the other "Customer database". The two databases have different owners. One of the generic items describing the filing cabinet is 'Drawer *n* holds an instance of the database called *d* '. Such information items are implemented as a label attached to a drawer. Another generic item is 'Drawer *m* holds instance *i* ', implemented by file cards in a drawer.

Example 10.2
A transaction processing system that provides access to several databases can also be described this way. Just replace Drawer by System Location.

Point 15
Some choices for a class of values may appear extreme but can be entirely practical.

However, this case can sometimes lead to difficulties as we will see in the next example.

Example 11
A database is required to hold archives of itself, so a variable is restricted to instances of "this" database.

Each time the database is archived an information item is inserted into the database; this information item holds the previous instance of the database. But the previous instance holds all earlier archived instances. Each archive is replicated again and again, which is unlikely to be the desired behaviour of the database. Even worse, the definition of permitted values includes the case where an information item holds itself. It is not clear whether this is possible nor whether the class of permitted values is well-defined.

Point 16
A description of a class of values that includes any kind of self reference should be viewed with suspicion and treated with great caution.

The database owner might still desire the database to hold instances of this database, either as a historical record or to provide backup instances in case of transaction failures. A safe way to do this is to remove potentially troublesome information items before storing the database instance. Thus what is stored is an instance of another database, one whose description is derived from this database by deleting certain generic items. Presumably the database designer will wish to apply the minimum safe deletion.

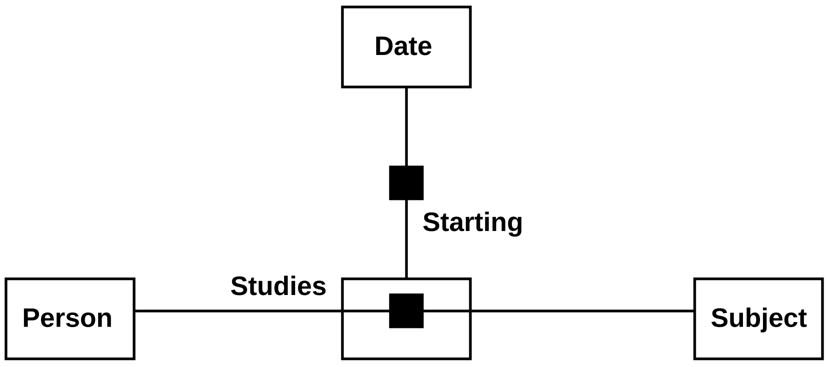
Point 17
There can be a need for derived conceptual data models. Database designers will need to know safe derivation rules.

For the next case we will move from values that are database instances to values that are individual information items.

Example 12
A variable is restricted to the permitted information items described by a particular generic item. In other words, the variable is restricted to a Fact Type defined in the data model.

Several styles of conceptual data modelling include this possibility in their notations. Figure 3.2.3.1 below contains an example drawn in the NIAM style, "UMIST" dialect. Recall that each small square stands for a distinct generic item. For each small square there is a corresponding Fact Type : all those information items formed by substituting permitted values in the generic item's variables. A rectangle surrounding a square stands for this Fact Type and can be used like any other rectangle. Recall also that such a Fact Type is said to be objectified or nested.

Figure 3.2.3.1 Information items as values



The picture in Figure 3.2.3.1 introduces two generic items :

Example 12.1
`'`*`p`*` studies `*`s`*`',`
`'`*`e`*` starting on `*`d`*`',`
where *e* is restricted to the permitted information items of '*p* studies *s*'.

Two permitted information items are :
```
'Carol studies Physics',
' 'Carol studies Physics' starting on 1.11.96'.
```

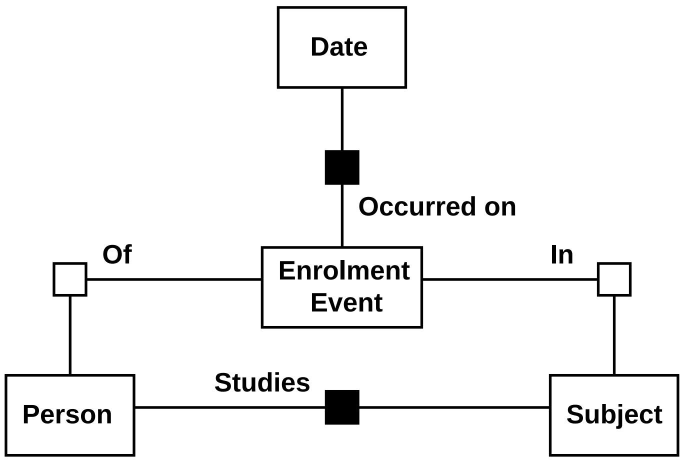
In NIAM this is defined to mean
```
'Carol studies Physics',
'This enrolment occurred on 1.11.96'.
```

Thus an alternative description is that 'Carol studies Physics' uniquely determines an enrolment event, and that this event occurred on 1.11.96. Thus we can translate the original generic items into :

Example 12.2
```
'p studies s',
'Enrolment event e occurred on d',
'Enrolment event ep is of the person pe',
'Enrolment event es is for the subject se'.
```

The picture for this alternative description is shown in Figure 3.2.3.2. There would be rules ensuring that the set of recorded enrolment events is uniquely determined by the set of recorded enrolments in any legitimate database instance.

Figure 3.2.3.2 An alternative description



This transformation of a class of permitted information items into a class of objects identified by the information items can always be done, and undone. One reason for preferring Figure 3.2.3.1 is that the picture is less cluttered, especially when constraint symbols are added. Another reason is that it will sometimes be difficult to think of a reasonable name for the class of objects.

Point 18
Any data model can be translated into an equivalent data model containing no nested Fact Types. However, nesting is a natural and useful feature of NIAM that should be retained in any model of NIAM conceptual data models.

Note the assumption we make about Figure 3.2.3.2 that an enrolment event cannot be recorded unless an enrolment determining the event is also recorded. The NIAM-style notations incorporate the equivalent assumption about Figure 3.2.3.1. It is an implicit rule. If enrolment events that were expected but did not happen also need to be recorded then an enrolment event is still identified by a person and a subject, but in some cases that person is not recorded as having enrolled in that subject. The white squares in Figure 3.2.3.2 would have to be shown as black and the construction in Figure 3.2.3.1 cannot be used to describe the database.

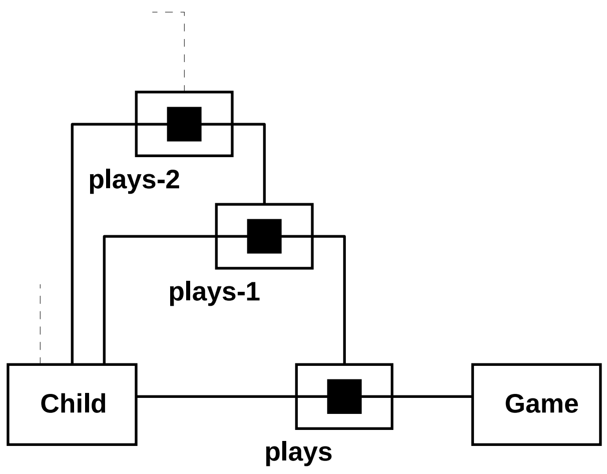
Now consider an extreme example of nested Fact Types.

Example 12.3
A database contains this sequence of information items :

```
'Tommy plays Football'
'Tommy plays 'Harry plays Football' '
'Tommy plays 'Harry plays 'Mary plays Football' ' '
```

If this seems unlikely think of someone playing a video recording of someone playing a video recording. These information items can be described by the conceptual data model in Figure 3.2.3.3.

Figure 3.2.3.3 Tommy plays games



The database owner says that the sequence of information items could be much longer than illustrated in Example 12.3. The database designers extend Figure 3.2.3.3 to show more generic items, stopping when it describes information items too big to be held in the physical database. The database owner says that when the physical database is full up he will move it to a bigger computer or to a bigger warehouse.

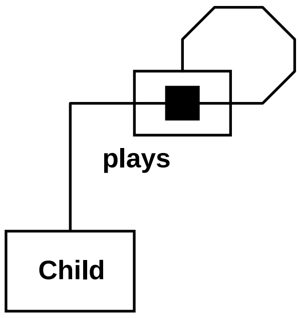
What should the database designers do now? They could use a completely different way to describe the information in the database. They could arrange to enhance the conceptual data model every time the database is moved. Or they could describe the database by an infinite sequence of generic items, so declaring from the start that there is no upper bound. Each would be a reasonable choice. The last has the merit of making things clear from the beginning. Unfortunately, an infinite number of generic items cannot be written down individually so the last case cannot make use of any of the NIAM-style notations. The generic items would have to be defined intensionally, not extensionally, for instance by a recursive definition.

Point 19
There are plausible, if unusual, database descriptions that cannot be represented in the conventional NIAM notations. Database designers will need to know how to recognise such cases.

Unfortunately, it is possible to be too extreme.

Example 12.4
Someone decides to go beyond Example 12.3 and draws Figure 3.2.3.4. An incomplete version of one of its information items is perhaps :
`'Tommy plays 'Harry plays 'Mary plays 'Betty plays ' ... ' ' ' ' '`

Figure 3.2.3.4 Tommy plays what ?



Here is another case of suspicious self reference. Figure 3.2.3.4 describes a generic item, *G*, whose second variable is restricted to information items described by *G*. Even if such information items were thought to exist they would be impractical. There is no useful encoding scheme that can compress every possible infinite sequence of names into a finite bit string.

Point 20
There are constructions in the conventional NIAM notations that should not be used. They either describe impractical information items or describe none at all. Database designers will need to know how to recognise such cases. Furthermore, recognition should not require ingenuity.

For the final case we will investigate a data model that has difficulties that must be overcome.

Example 13
The database holds the description of a data model. Perhaps it is part of a CASE tool.

Let us prescribe this design database in advance. We will start by writing down the most important generic items. These are the ones that describe the essentials of a core data model. The key generic items are :

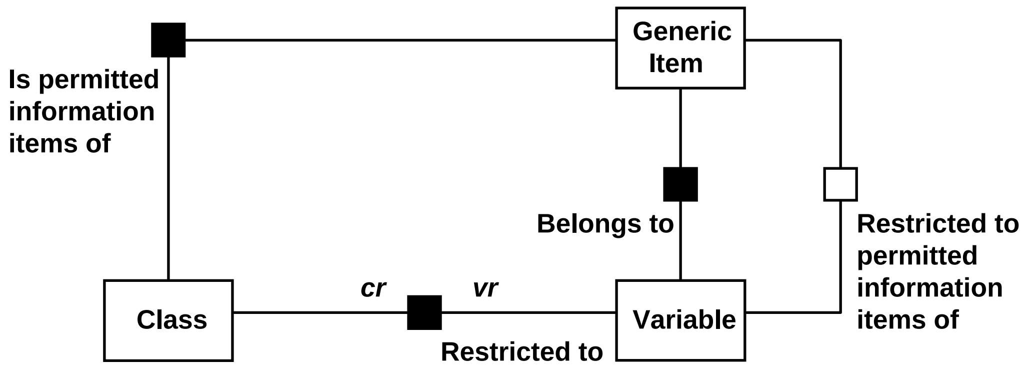
Example 13.1
'*v* is a variable of the generic item *g*'
'Variable *vr* is restricted to the class *cr*'
'Class *cp* is the permitted information items of the generic item *gp*'

There are some useful information items whose presence is uniquely determined in any legitimate database instance. One of them is :

Example 13.2
'Variable *vdp* is restricted to the permitted information items of the generic item *gdp*'

The generic items have been phrased in a way that makes it obvious which class each variable is restricted to. We can now draw the initial version of the conceptual data model, Figure 3.2.3.5 below.

Figure 3.2.3.5 Description of a database holding a conceptual data model (incomplete)


Generic Item
Is permitted information items of
Belongs to
Restricted to permitted information items of
Class
cr
vr
Variable
Restricted to

The picture you see in Figure 3.2.3.5 is an initial description of our design database. It is also an instance of a database : the picture holds information that we wish to be reminded of; it is possible to retrieve this information; and the information content can evolve to or from other instances. Indeed, it is an instance of the database we are describing, implemented on paper as it happens.

Unfortunately there is a suspicious self reference in Figure 3.2.3.5. In our description of the design database there is a small square that introduces the generic item :

```
'Variable vr is restricted to the class cr'.
```

In more detail, using a grammatic variant, this is

```
'Only members of the class cr are substituted for the variable vr in
legitimate database instances'.
```

Now Figure 3.2.3.5 is a database instance, and, by the rules of the notation, one of the items of information it contains is :

```
'Variable cr is restricted to the class Class'.
```

or in more detail :

```
'Only members of the class Class are substituted for the variable cr
in legitimate database instances'.
```

In this information item the class Class has been substituted for the variable *cr* in the generic item that describes the information item. If the information item is to be believed then a **member** of Class has been substituted for *cr*. Thus Class is a member of Class, making Class a very dubious conceptual object.

There is worse. Whatever the membership of Class might be it would be perverse to exclude any definable subclass of a member. One of these subclasses is

$$B =_d \{ b : \text{Class} \mid b \notin b \}.$$

We have Russell's paradox if we insist on this being a member of Class. (If $B \in \text{Class}$ & $B \notin B$ then $B \in B$; if $B \in \text{Class}$ & $B \in B$ then $B \notin B$). We must conclude that Class has a peculiar and inconvenient membership, or Figure 3.2.3.5 does not contain a legitimate database instance, or Class is not what it seems to be.

We do not want database designers to have to exercise great ingenuity in their choice of classes of values. Figure 3.2.3.5 seems to be a perfectly reasonable description of a useful class of objects. Perhaps Class is not what it seems to be.

Suppose that Class is a class of representatives, with each representative impersonating an object of interest to the database owner. This is much like your proxy voting for you at a shareholders meeting. Amend the awkward generic item so that it becomes

```
      'Variable vr is restricted to the class represented by cr'.
```

As an example, suppose we declare that the set, Nat, of natural numbers is used to represent classes. One information item could be

```
      'Variable p is restricted to the class represented by 5'
```

where 5 represents the class of all pirates. The awkward information item could now be

```
      'Variable cr is restricted to the class represented by 0'
```

where 0 represents the class, Nat, of representatives. In this information item the value 0 has been substituted for the variable *cr*. If the information item is to be believed then a member of Nat should be substituted for *cr*, which it has. There is no longer a paradox here.

Note that Nat does not have enough members to represent all conceivable classes, but it does have enough to represent each class that can be described separately in writing, which is all that we want in practice.

Point 21
Sometimes the class a variable is restricted to must be a class of representatives, each of which represents an object of interest to the database owner. Representatives allow paradoxes to be circumvented.

Representatives act as abstract identifiers. Why not use concrete identifiers, such as character strings, instead? One reason is to improve readability. The database owner and users will expect a concrete identifier to identify something. It is better if that something appears in the database description even if it is technically a representative. Another reason is that concrete identifiers belong to the actual database. In the early stages of a database design it can be inappropriate to include details of the actual database, as in Figure 3.2.3.5. Note that the exact class used for representatives does not matter very much. Information in the actual database will do all the identifying needed by the database users. The choice of class can be treated as an implementation decision that is deferred for ever.

We could declare that every class used in a conceptual data model is a class of representatives. Most objects used in practice could then be represented by themselves, just as you may vote on behalf of yourself at a shareholders meeting.

3.2.4 Preliminary conclusions

To sum up :

Point 22

a) Some databases need to be prescribed in advance.

b) On the face of it, the NIAM notation appears capable of prescribing what needs to be prescribed : a class of database instances declared to be legitimate, legitimate transitions between instances, and the meaning to be attributed to legitimate instances.

c) Some plausible database descriptions cannot be expressed in the notation. We should characterise these if possible.

d) Some constructions in the notation are improper. We should characterise these if possible.

e) Conceptual data models are evolving objects, and there can be a need for derived data models. We should characterise proper operations on data models if possible.

3.2.5 Requirements of any model of conceptual data models

We can now say what is required of any model of NIAM conceptual data models. Recall from Section 3.1 that an instance of a database is modelled as a set of tuples, and that each tuple is modelled as a family whose index set models the variables of a generic item.

Remember also that a database may contain statements about database instances and about conceptual data models, statements that are to be believed. The NIAM notations for conceptual data models allow us to introduce dubious constructions and paradoxes. We should choose a modelling tool that will help us to recognise or avoid these. We will choose pure set theory (ZF or VNB, whichever is more convenient from time to time).

In any proper NIAM data model each variable of each generic item is restricted to a fixed class of "permitted" values. We will model any such class by a set, never by a proper class. This avoids awkward circumlocutions and nothing useful is lost thereby. Thus for each generic item there will be a set of "permitted" tuples. It is a cartesian product, of course. By a modelling convention, the generic items of a proper data model give rise to pairwise disjoint cartesian products.

The model of any proper NIAM data model must uniquely determine a set of "permitted" tuples, and the legitimate subsets, alias legitimate instances, of this set. The components used to build the model must include sets modelling the variables of generic items, sets modelling the classes of permitted values, sets of Wffs modelling constraints, and sets modelling various kinds of annotation.

3.2.6 Credits

The receipt form in Example 4 has been adapted from a Cash Receipt Book (Duplicate With Carbon) by Silvine, product reference 228.

There are four ideas in NIAM that have been described here (Nijssen & Halpin [1989]; 2nd edition : Halpin [1995]). The first is that a database description should be understood by non-specialists with little assistance from experts. The second is using a description of the organisation to produce a description of the database. (This idea is not unique to

NIAM, of course). The third is the classification of many common kinds of database rule. The fourth is that information items can be useful values in information items.

The "UMIST" dialect of the NIAM-style notation has been adapted from Loucopoulos [1993] and Layzell & Loucopoulos [1989].

3.3 Introduction to Feature Notation

This is a brief introduction to Scheurer's Feature Notation, used to define complex set-theoretical models. A full treatment is given in Scheurer [1994], p410-431, with examples in p433 onward.

The final subsection covers the related topic of Scheurer's "levels of variation" in complex models. (As presented in Scheurer [1996], Section 6).

3.3.1 The problem

Consider the Predicator Model that was described in section 2.1.2. An information structure *I* was defined to be the 4-tuple $I =_d (P, O, Sub, F)$. This is a conventional definition but the convention has weaknesses when it is used as a modelling tool.

First, there are restrictions on the permitted values of *P*, *O*, *Sub*, and *F*; for instance, they belong to certain domains. Their definition is spread over several pages. It would be useful to have a compact and precise definition of *I* and its components in one place for reference.

Second, we often need to consider more than one information structure. For instance, several different information structures might need to be merged to give an overall information structure. *I* has the component parts *P*, *O*, *Sub*, and *F*. Suppose we also have the information structures *I'*, *J*, *K*, and *K1*. How would we name their components? Perhaps *I'* has the components *P'*, *O'*, *Sub'*, and *F'*, but for *J*, *K*, and *K1* we must introduce new names somehow. In addition, for *I'* we should declare that *P'*, *O'*, *Sub'*, and *F'* are restricted in the same way that *P*, *O*, *Sub*, and *F* are (and that *P'* corresponds to *P*, *O'* to *O*, etc). And similar statements must be made for the components of *J*, *K*, and *K1*.

Third, *Sub* is a binary relation with a particular domain, codomain, and graph. *Sub* also has component parts. How should these be named? They can be referred to by Dom(*Sub*), Cod(*Sub*), and Gr(*Sub*), but now we have two styles of reference to the internal components of *I*.

Any non-trivial model is liable to have this kind of problem. Any model, however excellent, will be rendered less effective if its definitions and results are written in an irregular notation.

3.3.2 The notation

The purpose of Feature Notation is to provide a straightforward and systematic naming scheme for the components of complex objects, and to make a clear distinction between the names of classes and the names of variables ranging over them.

Let us convert the conventional definition $I =_d (P, O, Sub, F)$ into a Feature Notation definition. First of all, we recognise that *I* is a variable ranging over a class which we might as well call InformationStructure, alias InfoStruc for short. Our purpose is to define this class and to provide a naming system for the component parts of each of its members.

We say that any *I* : InfoStruc has features. The four features given above are named *IP*, *IO*, *ISub*, and *IF*. For the variable *J* : InfoStruc they are named *JP*, *JO*, *JSub*, and *JF*, and so on. *ISub* is a relation that also has features : a domain, codomain, and graph. Assume that we have declared that any relation *R* of this kind has the features *RDom*, *RCod*, and *RGr*. Then the features of *ISub* are named *ISubDom*, *ISubCod*, and *ISubGr*. In general, a feature of any object has a name formed by appending a suffix to the name of the object; a feature of a feature has a name formed by appending a further suffix; and so on.

IP, *IO*, *ISub*, and *IF* are variable features of *I* : they vary as *I* ranges over InfoStruc. They are also primary features of *I*; once they are fixed then *I* is uniquely determined, but if any one of them is not fixed then *I* is not entirely determined. A secondary feature is any feature, such as *ISubDef* (the definition domain of *ISub*), that is uniquely determined by the primary features.

There are fixed features of *I*. These are the properties that are true for every member of InfoStruc. For instance, *IP* must be a set of predicators and the Base of each member of *IP* must be a member of *IO*, otherwise *I* would not be a member of InfoStruc. Fixed features are also given names. Their names are formed as usual by appending a suffix to the name of the variable. For instance, *ICond1* would be condition 1 for the variable *I*.

A Feature Notation definition of a class uses a standard layout like that shown below. There is a heading naming the class and introducing a variable belonging to that class. This is followed by a list of fixed and variable features, in any appropriate order, supplemented by comments. Primary variable features are distinguished by an asterisk. The symbol ".:" is a punctuation sign used to mark a feature as fixed. It is placed between the feature's name and its definition. A subscript "$_d$" means "by definition"; "$\subseteq_d$" means "is defined to be a subset of".

Now that *I* is defined in one place we can see that it has a rather awkward choice of primary features.

I : InfoStruc alias InformationStructure

Class of all conceptual data models (in the Predicator Model)

∗	*IO* : Set	The object type symbols occurring in *I*
∗	*IP* $\subseteq_d$ Predicators	The predicators occurring in *I*
	ICond1 .: $\forall p : IP \bullet$ Base(p) $\in IO$	Predicators are relevant
∗	*IF* $\subseteq_d$ *IO*	The Fact Type symbols occurring in *I*
	ICond2 .:	Fact Type symbols are sets of predicators, etc

IsFinite(*IF*) $\wedge$ IsPairwiseDisjoint(*IF*) $\wedge$ $\bigcup IF = IP$ $\wedge$
$\forall f : IF \bullet f \neq \varnothing \wedge$ IsFinite(f)

$IE =_d \{ x : IO \mid \text{IsETSymbol}(x) \}$	The Entity Type symbols occurring in *I*
$IL =_d \{ x : IO \mid \text{IsLTSymbol}(x) \}$	The Label Type symbols occurring in *I*
ICond3 .: $IO = IE + IL + IF$	Mutually exclusive and exhaustive
∗ $ISub : IO \leftrightarrow IO$	Subtype symbol relation Given any *x*, *y* : *IO* then *x ISub y* iff *x* is a subtype of *y*
ICond4 .:	*ISub* is a strict partial order with unique tops

$\text{IsTransitive}(ISub) \wedge \text{IsAntiReflexive}(ISub) \wedge$
$ISubGr \subseteq (IE{\times}IE + IL{\times}IL) \wedge$
$\forall x : IO \bullet x \in ISubDef \Rightarrow \exists 1\, t : IO \bullet x\ ISub\ t \wedge t \notin ISubDef$

This Feature Notation definition is equivalent to the following class definition in standard set-theoretical notation :

InfoStruc
$=_d \{ I \mid \exists\, IO, IP, IF, ISub \bullet$
$I = (IO, IP, IF, ISub) \wedge$
$IP \subseteq \text{Predicators} \wedge IF \subseteq IO \wedge ISub \in IO \leftrightarrow IO \wedge$
$\text{ICond1} \wedge \text{ICond2} \wedge \text{ICond3} \wedge \text{ICond4} \}.$

Note that the classes defined in Feature Notation will often be proper classes, not sets.

It is common practice to build models in stages. For instance, in the Predicator Model a schema, *S*, is defined to be the 2-tuple $S =_d (I, C)$ where *I* is an information structure as before and *C* is a set of constraint formulas. It may be appropriate for *S* to be described as an object with features *SI* and *SC*, and hence to have the features of features *SIO*, *SIP*, *SIF*, and *SISub*. However, the intention may be to treat *S* as an object that is like *I* but with the extra feature *C*. Then its features would be *SO*, *SP*, *SF*, *SSub*, and *SC*. The naming scheme allows us to do this without repeating any definitions. If a name suffix is enclosed in square brackets then that part can be omitted when further suffixes are added. If the features of *S* are given as *S*[*I*] and *SC* then the features that would otherwise be written as *SIO*, *SIP*, etc, can be written as *SO*, *SP*, etc. In *S*[*I*], *I* is called dispensable, and *SI* is called a group feature. In practice, the class name is often used as the suffix, giving us the feature *S*[*InfoStruc*]. Thus the class Schema can be defined in Feature Notation by

---

S : Schema (Extension of InfoStruc)

Class of all conceptual data models with constraints (in the Predicator Model)

∗ *S*[*InfoStruc*] : InfoStruc	The information structure, with features *SO*, *SP*, *SSub*, *SF*, etc.
∗ $SC \subseteq_d$ Wffs	Constraint formulas

3.3.3 Levels of variation

Often in modelling we are able to choose whether to concentrate on an entire class or on one particular member. Let us use a very simple database to illustrate this. We will use the student's mark database from section 3.1, Example 20 :

Name	**Mark**
Jim	53
Carol	73
Ann	64

This is a particular, fixed, instance of the database. We will model it here as a fixed set, Inst, of couples. (This ignores the links back to a generic item that enable us to make sense of the data). Its definition is

$$\text{Inst} =_d \{ (\text{Jim}, 53), (\text{Carol}, 73), (\text{Ann}, 64) \}$$

This is what the database users are interested in : the particular marks obtained by particular students. However, the database designer is interested in all possible instances since the particular students and their marks are not known in advance. The designer knows that the students' names belong to the fixed set Names, and that their marks belong to the fixed set Marks. An instance, such as Inst, must be a subset of $\text{Names} \times \text{Marks}$, so the designer models each instance as a relation $I : \text{Names} \leftrightarrow \text{Marks}$. The designer has fixed the sets Names and Marks, and allows instances to vary over the set, $\text{Names} \leftrightarrow \text{Marks}$, of all possible instances.

Now consider the method designer who wishes to describe techniques for designing tiny databases like this one. This designer is interested in all possible domains and codomains. The use of the sets Names and Marks is just one of many cases. This designer defines the class, TinyDB, of all possible cases, where $\text{TinyDB} =_d \{ X \leftrightarrow Y \mid X \neq \varnothing \wedge Y \neq \varnothing \}$. This designer has fixed the class TinyDB and allows designs of tiny databases to vary over its members.

These three viewpoints can be described as three "levels of variation". In level 1, the method designer's viewpoint, the use of just one binary relation is fixed and there is a description of all the various database designs. At this point the domain and codomain are variable. In level 2, the database designer's viewpoint, the domain and codomain are fixed and there is a description of all the various database instances. In level 3, the database user's viewpoint, the instance is fixed and there is a description of all the various student's marks. To sum up, each level fixes one or more features that are variable in the previous level(s). This is illustrated in Table 3.3.3.1 below.

Table 3.3.3.1 Three levels of variation

Level	**Viewpoint**	**Fixed features**	**Variable features**
1	Method designer	Binary relations	$X, Y, X \leftrightarrow Y$
2	DB Designer	X, Y	I : X↔Y
3	DB User	I e.g Inst	(s, m) : IGr

It is important to recognise the different levels of variation in the model of a system. One benefit is to remind us that the operations we might wish to define are different in each level. For instance, in level 2 of our example we can define a change of database instance, but not a change of domain or codomain.

Another benefit is to remind us that a supposedly fixed, constant, object might really be an exemplar or generic object. If it is, then it should be recognised as a variable ranging over a certain class, and the class ought to be defined.

Note that the number of levels may be chosen for convenience. In Table 3.3.3.1, level 2 could be split into two levels. In the upper level the designer gives the requirements for the domain and codomain; in the lower level the implementers choose particular sets meeting those requirements. Likewise, there can also be a choice of definitions. In level 3, the feature fixed at this level could be a sequence of database instances, with the last one being the current instance.

4 Core model : Definition

In this chapter we will begin the job of building a set-theoretical model of "all" well-formed NIAM conceptual data models. The model will be used to help answer the questions posed in Section 1.1.

Remember from Chapter 3 that any conceptual data model consists of a core structural part supplemented by annotations of various kinds, such as names and marks (Section 3.2.2, point 9). We will begin by modelling the core structural part, and will call it the core model. This will be done in three stages. First, we will concentrate on the definition of the model, and on the properties of each model of a data model (this chapter). Next, we will concentrate on operations that model the conversion of one data model into another, and on some equivalence relations (Chapter 5). Finally, we will show that this model of "all" well-formed data models does indeed include all that it should (Chapter 6).

Before starting, we need to decide what requirements the model should satisfy and what principles it should conform to. We should also fix the terminology that we will be using when saying what is being modelled. (Section 4.1).

We can then define the model of "all" well-formed (core) data models. (Section 4.2). On its own, the definition is not very useful and not obviously correct. We must investigate its properties and prove that each model of a core data model does what Section 3.2.2, point 9, requires it to do. (Section 4.3). It is then appropriate to investigate a few more properties that illuminate the structure of core data models. They also help us to define preconditions for operations on data models. (Section 4.4).

If the model is to be useful then there must be a straightforward translation from each model of a data model into a diagram in any of the NIAM pictorial notations. There must also be straightforward translations into convenient representations inside design tools. (Section 4.5).

By their nature the models are very mathematical. Where possible, definitions and statements of properties have been split into two parts : a less formal description of items of interest to data modellers and tool designers, ("overview" or "outline"), and the mathematical details, ("details").

4.1 Considerations and plan of attack

4.1.1 Terminology

The description of NIAM given in Chapter 2 talks of facts, roles, entities, Fact Types, Entity Types, etc. Unfortunately, the definition of some of the terms varies from publication to publication. In addition, the description of databases and data models given in Chapter 3 talks of information items, generic items, variables, tuples, indexes, etc. We must fix our usage before attempting to do any modelling. We will use NIAM terms where possible, as follows.

Fact is a key term in NIAM but it is potentially confusing. A "fact" held in a database is not necessarily true, and we wish to talk of "facts" that are not currently held in the database. Moreover, it is not always clear whether a fact is a complete information item, such as 'Jack arrived on 1.11.96', or a tuple, such as (Jack, 1.11.96), holding just the variable part of the information item. We will use the word "fact" only as a qualifier where the kind of object referred to is clear.

Tuples are the units of information held in a database instance. Each tuple holds the variable part(s) of an information item, as described in Section 3.1. Such a tuple is always represented as a family with a designated index set.

A **Fact Type** is a cartesian product, and hence a set of tuples, that has been introduced by a conceptual data model.

A **role** is an index of certain tuples, one that has been designated as such by a conceptual data model. Role will be used as a synonym for tuple index; any other connotations will be ignored in the models.

An **entity** is any object, such as a person, number, car, or name, that has been designated by a conceptual data model as an element of certain tuples, but whose internal structure is not described by the data model. Thus an entity is an atomic object with respect to the data model it occurs in. The tuples of a Fact Type will not be called entities. Note that this does not prevent an entity itself being a tuple, such as a date or a complex number, described elsewhere.

An **Entity Type** is a set of entities designated by a conceptual data model.

A **Label Type** is an Entity Type highlighted in a conceptual data model as being one whose entities must be represented directly in the implementation of the actual database. Thus any **label** is also an entity. This highlighting is not part of a core data model.

Information items and **generic items** will not be modelled explicitly. In effect, they are abstracted away.

4.1.2 Considerations

According to Section 3.2.2, point 9, a core data model prescribes the delimiting class of information items within which the specified database must evolve. Thus, any proper core model must define a class of tuples. There is nothing to be gained by allowing this class to be a proper class; we will require it to be a set. A data model declares that the variable parts of information items are restricted to certain classes, such as people or names. Thus any proper core model must define a set of cartesian products; the delimiting set of tuples is the union of these cartesian products.

Each tuple will be represented as a family; that is, as a function from an index set to suitable values, alias elements. Indexes will be called roles to match the NIAM terminology. In some tuples each element of the tuple is a primitive object. Such primitive objects will be called entities to match the NIAM terminology. If all tuples were like this then any model of data models would be trivial. However, a data model can require that some tuples have elements that are tuples defined by the data model. Any model of data models should preserve this construction.

Our primary concern is database design, not database implementation and use. We are not solely concerned with completed conceptual data models. We are also interested in incomplete data models that arise at intermediate stages in the design process. In other words, we view a data model as an evolving object. It will be seen in the next section that the intermediate stages of interest are themselves proper data models. In particular, an initial empty data model, such as a blank sheet of paper or a blank screen, can be deemed to be a proper data model. Any model of data models should facilitate its description as an evolving object, and define an "empty" model to be well-formed.

We have a choice between modelling the NIAM notation explicitly or modelling the objects we believe a data modeller wishes to define when using NIAM. As the rules of the notation are not understood well enough (by anyone, apparently) we should prefer to avoid modelling the notation too directly. In effect, we should model an abstract notation that is similar to the NIAM notation.

Before the model is analysed we may need to use intuitions to guide us when saying what is a well-formed model and what is not. If there is a choice we should prefer a model that requires fewer intuitions and more elementary intuitions.

Initially, our definition of what constitutes a "well-formed" data model may be somewhat tentative, and we may wish to talk of "ill-formed" data models in order to discuss their deficiencies. Describing operations that alter data models may require us to describe partial data models that are clearly ill-formed. The description of these operations may be made simpler and clearer if one model covers both partial and well-formed data models. Thus we should prefer a model that enables us to describe the more reasonable ill-formed data models.

4.1.3 Plan

One possibility is to model each core data model as a set of cartesian products. This concentrates on the objective of any core data model and abstracts away everything else. This model has the advantage of being independent of any data modelling notation. There is no possibility of the notation restricting the model so that some databases cannot be described. We can be sure that it covers all databases. However, this generality is also a disadvantage. In effect, the model says that the database designer can describe the database in any way whatsoever, using any notation that can be implemented with pencil and paper or a very flexible word processor. With this description there is very little that a CASE tool could do to help the database designer. The model does nothing to tell us how a design tool could prevent improper database designs being produced or could help to manipulate design information.

We will not use sets of cartesian products as a model of core data models, but we will use them later on, in Chapter 6, to test the completeness of the NIAM notation.

The ultimate constituents of a core data model are entities and roles. Sets of roles form the index sets of tuples. Entities form the elements of tuples, and sets of entities are used to determine cartesian products. Tuples can also be elements of tuples, but these elements are themselves ultimately composed from entities and roles. No other constituents are apparent or necessary.

The main model of core data models, which is established in this and the next chapter, is constructed from sets modelling entities and roles, and nothing else. The model does not model the NIAM notation's symbols, but it does model the constructions that those symbols describe. That is, it models what the data modellers are saying, but not the symbols used to say it.

4.2 The definition of DaMod0

In this section we will define a set-theoretical model of "all" well-formed core data models. That they are well-formed will be plausible, but the proof of this is left until the next section. That it models "all" in some reasonable sense will also be plausible, but the proof is left until Chapter 6.

We will start with an example of the step-by-step construction of a conceptual data model. In parallel, we will build an ad hoc set-theoretical model that will clearly demonstrate that this particular data model is well-formed. We will then use this example to guide our definition of the model of core data models.

4.2.1 An example

Let us assume that we have been asked to design and implement a small database holding information about students. The requirements are not entirely clear but we know that the database must record the subjects studied by certain students and the marks they get in the exams for those subjects. We are told that the database contents will evolve as the population of students changes and as their exam marks become available.

We will draw a NIAM conceptual data model. Obviously, the first step is to allocate some space to draw it in : Figure 4.2.1.1.

Figure 4.2.1.1 A conceptual data model : Step 1

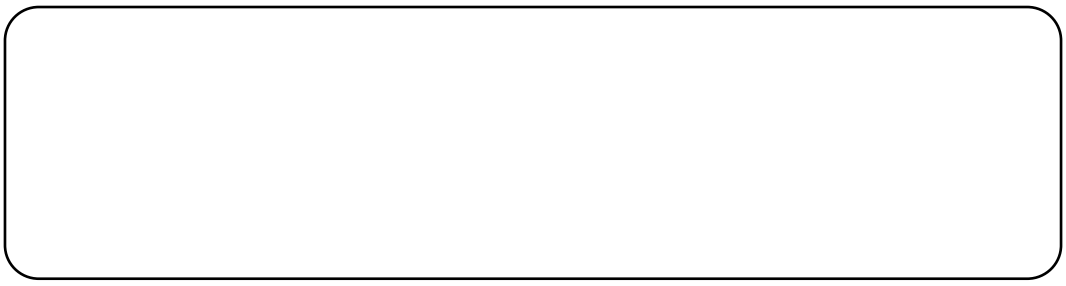
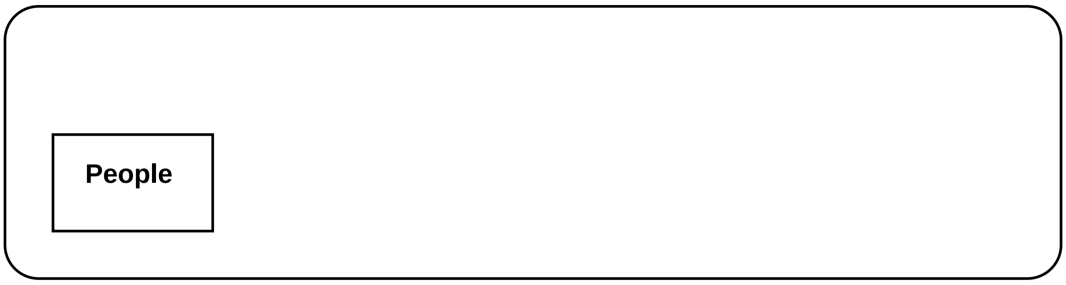


Figure 4.2.1.1 contains a (the) blank or empty data model. It describes no tuples at all. Any database conforming to this specification must evolve within the limits of the empty set of tuples and so will be permanently empty. Although this is a very uninteresting database, it is nevertheless a perfectly well defined one. We can, and will, deem the empty data model to be well-formed.

The database will hold facts about students, such as the students Carol, Jim, and Ann. We will assume that students are primitive objects whose internal structure will not be described by the data model. Thus each student is an entity. The population of students changes from year to year, whereas in NIAM we prefer to use domains that are fixed. However, we can say that every student is a person. We will introduce an Entity Type, which we will call People, consisting of every person, past, present, and future : Figure 4.2.1.2.

Figure 4.2.1.2 A conceptual data model : Step 2



We will model this Entity Type by the fixed set People. The exact membership of People does not matter a great deal provided it is large enough to model all people. We do require it to be non-empty, for two reasons. First, the members of People will be used as elements of tuples. If People were empty then its members would be the elements of no tuples, making its introduction rather pointless. Second, we wish to avoid the possibility of two supposedly distinct sets both being empty, and so not distinct after all. In this example we will assume that People contains at least three distinct members, Carol, Jim, and Ann, that model our three students Carol, Jim, and Ann. As explained in Section 3.2.5, we model in pure set theory so Carol, etc, are sets, but their memberships do no more than distinguish one from another.

Figure 4.2.1.2 does nothing to say that any database instances can contain tuples. Just like the empty data model, it specifies a database that is permanently empty. Unlike the empty data model it contains an Entity Type that is unused, so that we can say that the data model is obviously incomplete. Nevertheless, the database it specifies is well defined so we will deem this data model to be well-formed (though incomplete). We will not define "complete" in this work but we will note several circumstances in which a data model is clearly incomplete.

The database will hold facts about subjects, such as the subjects Physics, Mechanics, Maths II, and Maths III. We will assume that subjects are also primitive objects whose internal structure will not be described by the data model. We will introduce an Entity Type, which we will call Subjects, consisting of every possible subject, including those not known to us yet : Figure 4.2.1.3.

We will obey the NIAM notation rule that different Entity Type symbols stand for different objects. Thus the introduction of Subjects implicitly declares that People and Subjects are different sets of entities. Furthermore, we will obey the NIAM modelling rule that the different Entity Types in a data model are pairwise disjoint, so no subject is a person. This makes it easier to see which kinds of information can be recorded about which kinds of entity. For instance, if Physics were recorded as being a Science subject then we can be sure that this says nothing about a person. (Overlapping sets of entities can be described by using Subtype symbols, but these are constraint symbols and so not a feature of core data models).

Figure 4.2.1.3 A conceptual data model : Step 3

People
Subjects

We will model this second Entity Type by the fixed set Subjects. As before, the exact membership of Subjects does not matter a great deal provided it is large enough and non-empty, but we also require it to be disjoint from People. In this example we will assume that Subjects contains at least four distinct members, Physics, Mechanics, Maths2, and Maths3, that model our four subjects Physics, Mechanics, Maths II, and Maths III.

Once again, Figure 4.2.1.3 specifies a database that is permanently empty. The database it specifies is still well defined so we will also deem this data model to be well-formed (though incomplete).

We wish to record the subjects studied by certain students. Let us list some examples and see what they have in common. For instance, we might want to record

' Carol studies Physics ';
' Jim studies Physics ';
' Jim studies Maths II '.

We can see that these information items follow the pattern described by the generic item

' *p* studies *s* '

where the variable *p* is restricted to people and the variable *s* is restricted to subjects.

As is typical, this database will hold only the variable parts of information items, in the form of tuples. Each tuple will say which variable in the generic item is to be replaced by which value in order to reconstitute the full information item. It does this by providing a mapping from the generic item's variables to values, alias elements. For instance, the tuple

(p $\mapsto$ Carol, s $\mapsto$ Physics)

encodes the information item

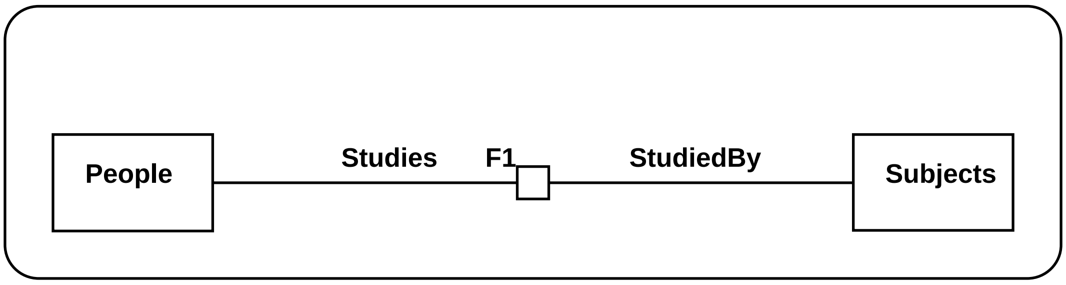
' Carol studies Physics '.

Note that when used inside a tuple the generic item's variables are no more than symbols and are properly shown as constants. From now on we will call them indexes, alias roles.

Thus we wish to introduce a Fact Type, a cartesian product, into our data model. The Fact Type consists of all those tuples where the role p maps to a person and the role s maps to a subject. These are the tuples the database is permitted to hold. Each database instance will hold some subset of these tuples.

We will introduce this Fact Type indirectly. The data model will declare that there are two roles used together as the index set of tuples, and that one role is associated with the Entity Type People and the other with the Entity Type Subjects. To make the data model easier to read we will give the role p the alternative name Studies and s the alternative name StudiedBy. The data model now provides enough information for us to derive the contents of the Fact Type : Figure 4.2.1.4.

Figure 4.2.1.4 A conceptual data model : Step 4



In our ad hoc set-theoretical model we will model the roles Studies and StudiedBy by the two sets Studies and StudiedBy. It does not matter which sets these are as they are arbitrary indexes. As with the sets modelling entities, the memberships of the sets modelling roles do no more than distinguish one from another. Remember that the two Entity Types in the data model were modelled by the sets People and Subjects. The association of roles with Entity Types declared in the data model is obviously modelled by the mapping

$$(\text{Studies} \mapsto \text{People}, \text{StudiedBy} \mapsto \text{Subjects}).$$

From this we can derive the cartesian product

$$\text{F1} =_d \{ \ (\text{Studies} \mapsto x, \text{StudiedBy} \mapsto y) \mid x : \text{People} \ \wedge \ y : \text{Subjects} \}$$

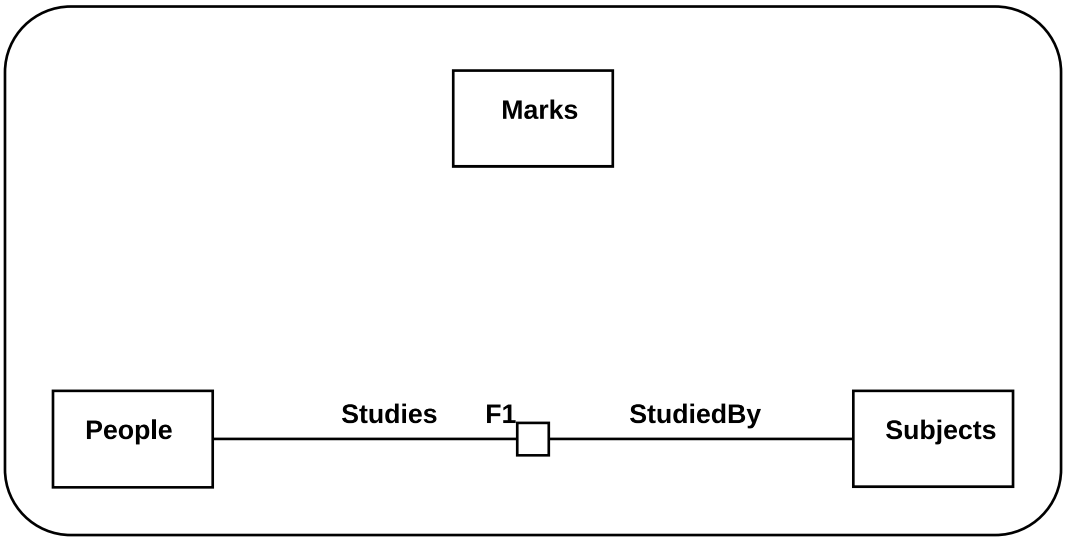
which models the Fact Type introduced in this step of the design.

F1 is obviously well defined; that is, F1 is uniquely determined given Studies, StudiedBy, People, and Subjects. Thus any database that is modelled as an evolving subset of F1 is obviously also well defined. This is so regardless of the particular sets, People, Subjects, Studied, and StudiedBy, that we happened to choose. Thus we can be confident that the data model in Figure 4.2.1.4 is well-formed; that is, the data model can be given, and has been given, a well defined meaning.

This data model is also incomplete but in a different way. The database is not permitted to record exam marks and so does not meet all the requirements. This kind of incompleteness cannot be detected by inspecting the data model in isolation.

As just noted, the database will hold facts about exam marks. At the moment we do not know whether 105 is a permitted mark, or 5½, or B+. We will leave the definition of permitted marks as an implementation decision, which might be different for different implementations. Consequently, we declare that marks are primitive objects whose internal structure will not be described by the data model but we do not say which objects yet. We will introduce an Entity Type, which we will call Marks, consisting of every permitted mark : Figure 4.2.1.5. To keep the example simple we will now model one particular implementation choice.

Figure 4.2.1.5 A conceptual data model : Step 5



We will model this third Entity Type by the fixed set Marks. As usual, the exact membership of Marks does not matter a great deal provided it is the right size and non-empty, and provided that the sets People, Subjects, and Marks are pairwise disjoint.

Figure 4.2.1.5 introduces the same Fact Type(s) as did the previous step. The database is still modelled as an evolving subset of F1, which is well defined. We will deem this data model to be well-formed (though incomplete).

We wish to record the exam marks obtained by certain students. For instance, we might want to record that

' Carol got 73 in Physics '.

However, we know from the requirements that a student can get an exam mark only in a subject studied by that student. This is a constraint and constraints are not a feature of core data models, but we can take advantage of it. We can say that something got 73 and that this something (a studyship?) is uniquely identified by the information item

' Carol studies Physics '.

Thus some example information items, phrased in an unusual way, would be

' ' Carol studies Physics ' got 73 in the exam ';

' ' Jim studies Physics ' got 53 in the exam '.

We can see that these information items follow the pattern described by the generic item

' *g* got *m* in the exam '

where the variable *m* is restricted to marks and the variable *g* is restricted to permitted information items described by the generic item ' *p* studies *s* '.

The database will hold these information items in the form of tuples such as

(g $\mapsto$ (p $\mapsto$ Carol, s $\mapsto$ Physics), m $\mapsto$ 73)

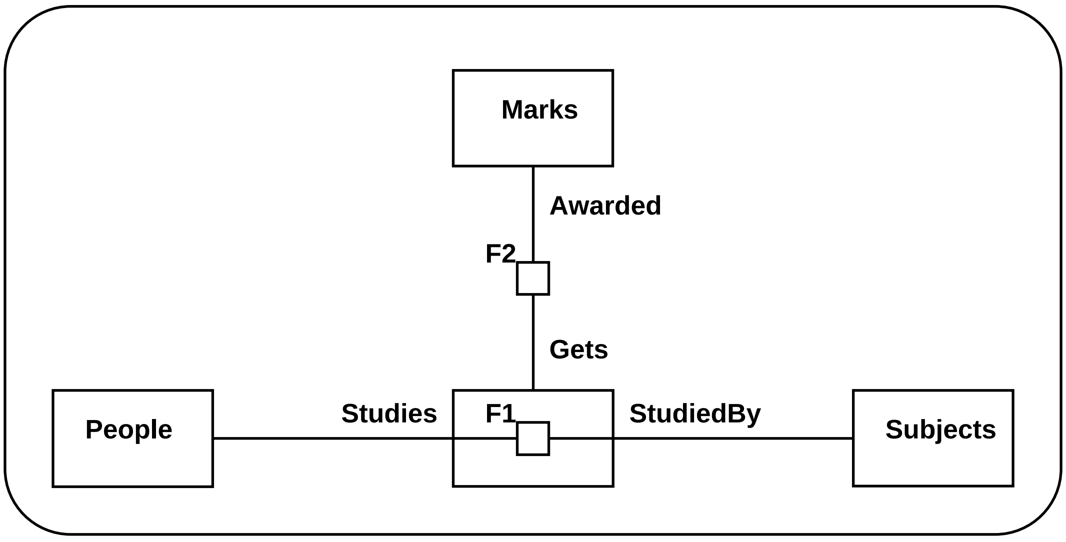
where one of the elements is a tuple already introduced by the data model. Thus we wish to introduce a Fact Type consisting of all those tuples where the role g maps to a tuple belonging to the Fact Type whose roles are p and s, and the role m maps to a mark. Database instances are now restricted to tuples drawn from two Fact Types.

For convenience, whenever we introduce more roles into a data model we will choose roles that are different from those already in use. This avoids us having to use awkward phrases such as "the role x when used in this Fact Type rather than in that one". Since roles represent arbitrary placeholders in generic items this choice does not restrict our ability to specify databases. Thus we have chosen roles p, s, g, m that are distinct.

A by-product of this rule is that the Fact Types introduced by a data model are pairwise disjoint : the tuples of different Fact Types use different roles as indexes. If we form the union of all the Fact Types introduced by a data model we can be sure that each tuple in the union belongs to exactly one of the Fact Types. We can therefore describe a database as an evolving subset of this union without losing any information; we do not have to commit ourselves to any particular implementation organisation.

As before, we will introduce this Fact Type indirectly. The data model will declare that the roles g and m are used in a Fact Type and that g is associated with the Fact Type whose roles are Studies and StudiedBy, and that m is associated with the Entity Type Marks. To make the data model easier to read we will give the role g the alternative name Gets and m the alternative name Awarded : Figure 4.2.1.6.

Figure 4.2.1.6 A conceptual data model : Step 6



In our ad hoc set-theoretical model we will model the roles Gets and Awarded by the two sets Gets and Awarded, distinct from Studies and StudiedBy. The association of the roles Gets and Awarded with domains (Entity Type or Fact Type) is obviously modelled by the mapping

$$(\text{Gets} \mapsto \text{F1}, \text{Awarded} \mapsto \text{Marks})$$

where F1 is the cartesian product defined earlier. From this we can derive the cartesian product

$$\text{F2} =_d \{ \ (\text{Gets} \mapsto x, \text{Awarded} \mapsto y) \ | \ x : \text{F1} \ \wedge \ y : \text{Marks} \}$$

which models the Fact Type introduced in this step of the design.

F2 is obviously well defined so any database that is modelled as an evolving subset of F1 $\cup$ F2 is obviously also well defined; thus Figure 4.2.1.6 is also well-formed. We have made Fact Types pairwise disjoint so from now on we will use "+", as in F1 + F2, to signal the union of disjoint sets.

Now the future database users tell us that exam marks must be checked before they can be considered final and that they want to note these checks in the database. We therefore need to record information items such as

' That Carol got 73 in Physics has been checked '.

Clearly a result should be recorded as checked only if the result itself has been recorded. We can rephrase this information item in terms of another recordable information item :

' That ' ' Carol studies Physics ' got 73 in the exam ' has been checked ',

leading us to define the generic item

' That *c* has been checked '

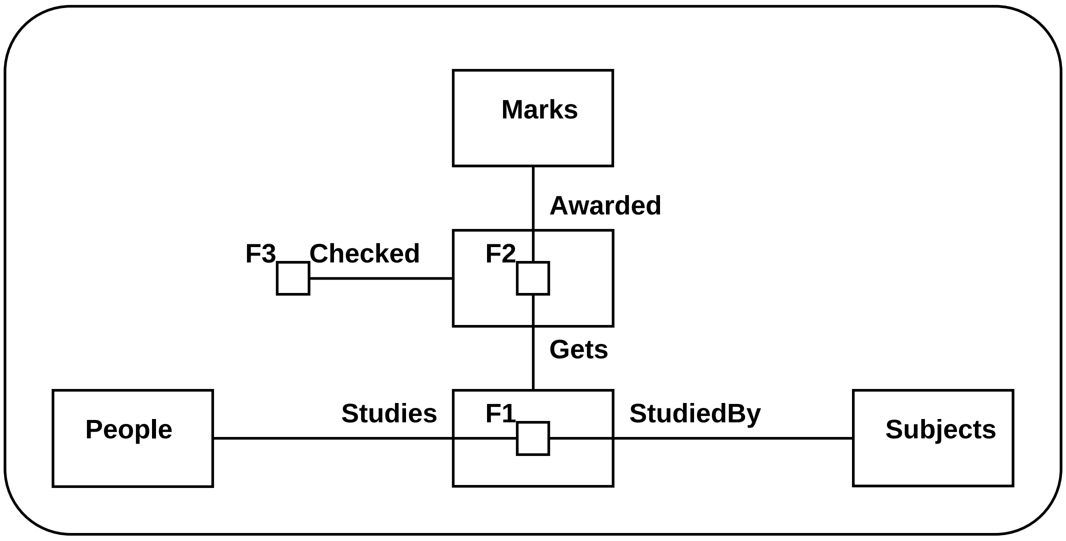
where the variable *c* is restricted to permitted information items described by the generic item ' *g* got *m* in the exam '.

The database will hold this kind of information item in the form of tuples such as

(c $\mapsto$ (g $\mapsto$ (p $\mapsto$ Carol, s $\mapsto$ Physics), m $\mapsto$ 73)).

Thus we wish to introduce a Fact Type consisting of all those tuples where the role c, given the alternative name Checked, maps to a tuple belonging to the Fact Type whose roles are g and m : Figure 4.2.1.7.

Figure 4.2.1.7 A conceptual data model : Step 7



We will model the role Checked by the set Checked, and the association of the role with its domain (a Fact Type here) by the mapping

(Checked $\mapsto$ F2).

From this we can derive the cartesian product

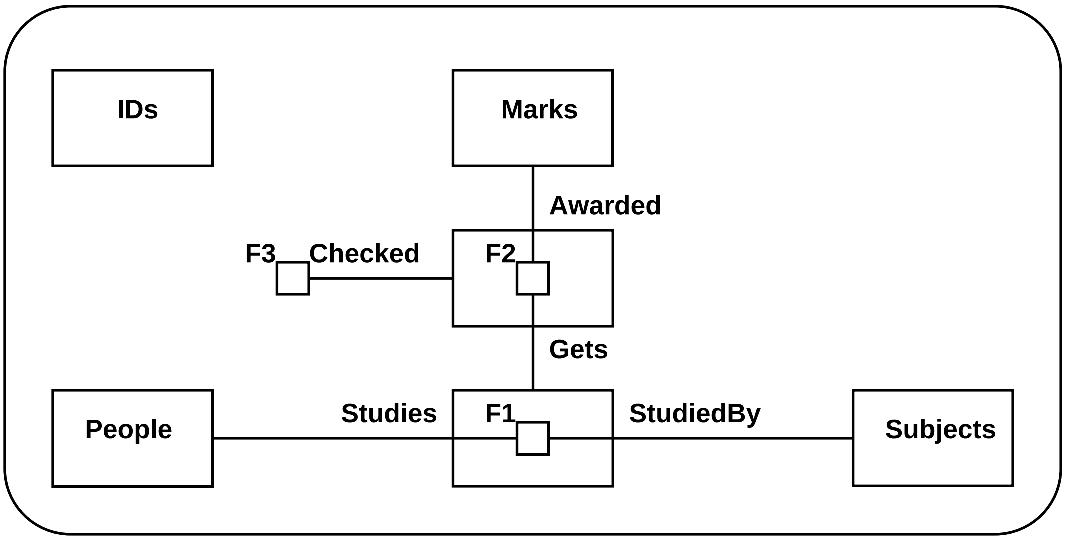
F3 $=_d$ { (Checked $\mapsto$ *x*) | *x* : F2 }

which models the Fact Type introduced in this step of the design. Notice that F3 is a unary cartesian product, a natural way to model ticks on a list of things to be done.

F3 is obviously well defined so the union F1 + F2 + F3 is obviously also well defined; thus Figure 4.2.1.7 is well-formed.

As it stands, the conceptual data model in Figure 4.2.1.7 specifies a conceptual database holding (conceptual) tuples composed from people, subjects, and marks. We must ensure that the data model specifies some practical tuples that can be held in the actual database. (Remember from Section 3.1.2, point 10, that the actual database is the implemented part of a conceptual database). We will use some kind of identifier to refer to each student. For instance, Carol's identifier might be 5173, Jim's 5926, and Ann's 6018. We will introduce an Entity Type, which we will call IDs, consisting of every possible student identifier : Figure 4.2.1.8.

Figure 4.2.1.8 A conceptual data model : Step 8



We will model this Entity Type by the non-empty fixed set IDs, disjoint from People, Subjects, and Marks. Figure 4.2.1.8 introduces the same Fact Types as did the previous step. We will deem this data model to be well-formed (though incomplete).

We will declare that the conceptual database holds information items recording which current student has which identifier, for instance

' Carol has the student identifier 5173 '

' Jim has the student identifier 5926 '

' Ann has the student identifier 6018 '.

These follow the pattern described by the generic item

' *pv* has the student identifier *iv* '

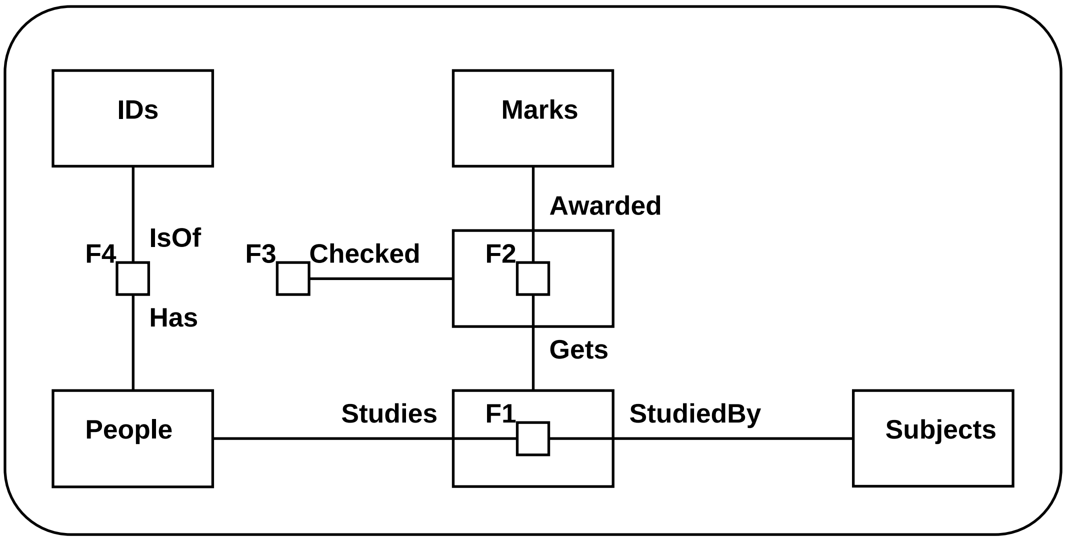
where the variable *pv* is restricted to people and the variable *iv* is restricted to student identifiers. The database will hold this kind of information in the form of tuples such as

(pv $\mapsto$ Carol, iv $\mapsto$ 5173).

Note that these tuples could be implemented physically by labels pinned to students, but here we decide that such tuples are outside the actual database that is to be accessed and updated by our users. The decision would be recorded by marking the Entity Type IDs as a Label Type, and People as not a Label Type. Such markings are not part of the core data model and so are left until Chapter 7.

We will introduce a Fact Type into our data model consisting of all those tuples where the role pv maps to a person and the role iv maps to a student identifier. We will give pv the alternative name Has and iv the alternative name IsOf : Figure 4.2.1.9.

Figure 4.2.1.9 A conceptual data model : Step 9



We will model the roles Has and IsOf by the two sets Has and IsOf. The association of roles with their domains is modelled by the mapping

$$(\text{Has} \mapsto \text{People}, \text{IsOf} \mapsto \text{IDs}).$$

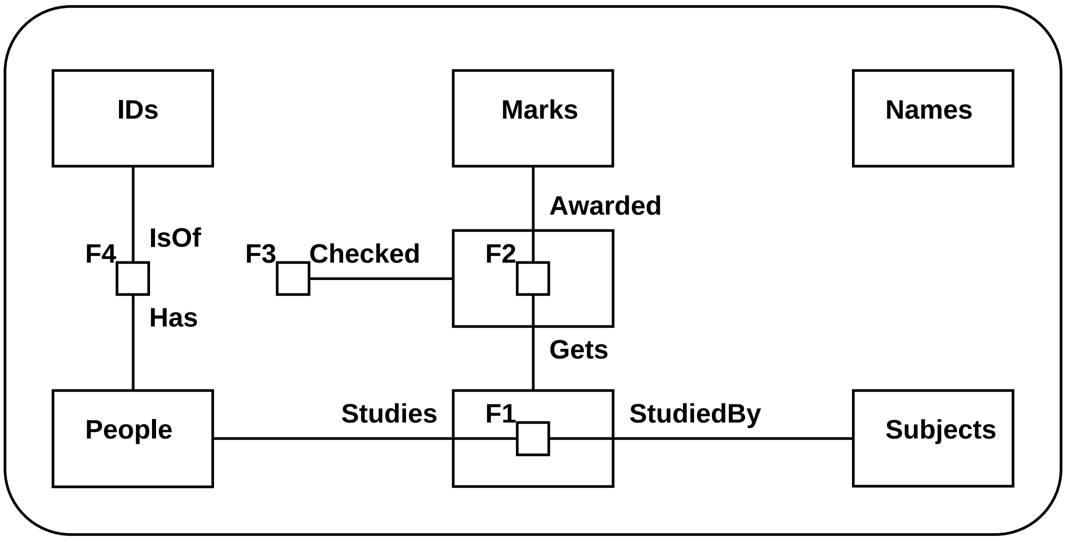
From this we can derive the cartesian product

$$\text{F4} =_d \{ \ (\text{Has} \mapsto x, \text{IsOf} \mapsto y) \ | \ x : \text{People} \ \wedge \ y : \text{IDs} \}.$$

F4 is obviously well defined so F1 + F2 + F3 + F4 is obviously also well defined and Figure 4.2.1.9 is well-formed.

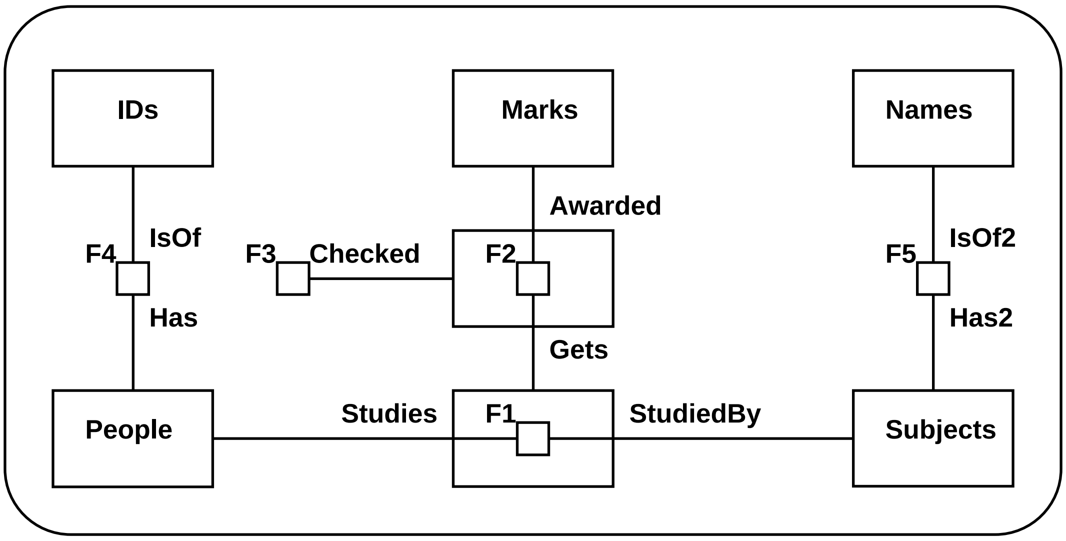
We need to do the same for subjects. We will introduce an Entity Type called Names, modelled by the set Names : Figure 4.2.1.10.

Figure 4.2.1.10 A conceptual data model : Step 10



And we introduce a Fact Type whose roles are Has2 and IsOf2, modelled by the sets Has2 and IsOf2 : Figure 4.2.1.11.

Figure 4.2.1.11 A conceptual data model : Step 11



This Fact Type is modelled by the cartesian product

$$F5 =_d \{ (\text{Has2} \mapsto x, \text{IsOf2} \mapsto y) \mid x : \text{Subjects} \wedge y : \text{Names} \}.$$

F5 is obviously well defined so any database that is modelled as an evolving subset of F1 + F2 + F3 + F4 + F5 is obviously also well defined. This is so regardless of the particular sets that we happened to choose in our ad hoc model. Thus we can be confident that the data model in Figure 4.2.1.11 is well-formed.

Notice that subject names are character strings. Although each name is defined to be a primitive object with respect to the data model it has an internal structure that is defined elsewhere. The Entity Type Names is, in fact, a set of n-tuples, (n : Nat).

We have now finished our conceptual data model. The ad hoc set-theoretical model has produced a well defined set of cartesian products from which we conclude that the data model is well-formed. That is, for each small square in the data model there will be a single well defined non-empty set of tuples once certain implementation decisions have been taken (provided they are reasonable decisions, of course). The database must be capable of storing certain subsets of these tuples, and is not permitted to store any other tuples. Note, though, that being well-formed does not guarantee that the database will meet the users' requirements. For instance, the database as specified here cannot record students' names. This might or might not be what is wanted.

The data model in Figure 4.2.1.11 is incomplete in two different ways. First, the data model does not introduce any tuples belonging to the actual database; for instance, there are no tuples connecting people identifiers to subject names. This is normal NIAM practice; the definition of the actual database is left until the implementation phase of the project. Second, Figure 4.2.1.11 is a core data model. It lacks annotation defining constraints on the database contents. It lacks markers indicating which part of the conceptual database belongs to the actual database. In principle, it also lacks all text annotation such as Entity Type names. The text appearing in the figures is there to help us talk about the example; it is not part of the core data models.

We will finish this example by displaying an instance of an implementation of the database : Figure 4.2.1.12. Perhaps the implementers used the algorithm given in Halpin

[1995], p251-291 to translate Figure 4.2.1.11 into a relational database schema. In effect the algorithm adds the missing Fact Types to the data model, such as one connecting people identifiers to subject names. It also takes advantage of the fact that each person-subject combination gets at most one mark, which in turn has at most one tick, to pack up to three tuples from different Fact Types into each row of the table. Notice that the implementers have changed the unary Fact Type whose only role is Checked into a binary Fact Type whose other domain is the singleton Entity Type {Yes}. This kind of implementation change is normal NIAM practice.

Figure 4.2.1.12 An instance of the real database

Student ID	Subject	Mark	Checked
5173	Maths III	68	
5173	Mechanics		
5173	Physics	73	Yes
5926	Maths II	57	Yes
5926	Physics	53	
6018	Maths III		
6018	Physics	64	Yes

4.2.2 Definition overview

In the example we showed that we had a well-formed though possibly incomplete data model at each step, and concluded that the final step was well-formed. Each step introduced one new element which was obviously well defined provided the elements introduced earlier were well defined, which they were. Clearly, experienced data modellers would be confident that a well-formed data model was being constructed without the need to produce a written proof. Even if they were shown only the final step they would be able to imagine a step by step construction that would convince them that the data model is well-formed. Therefore, to this extent, the NIAM design technique as described so far is capable of producing well defined database specifications.

However, we may wonder if there are well-formed data models that cannot be constructed in a step by step fashion (though we will see later on that there are none), and it may not be intuitively obvious to students and inexperienced data modellers whether a data model is well-formed or not. Some simple rules or tests for well-formedness are desirable. Furthermore, it is desirable that a design tool used to construct data models is able to prevent the construction of ill-formed data models. It, too, should implement simple rules if possible. In other words, there is still a need to build a model of "all" well-formed data models so that we can investigate its properties.

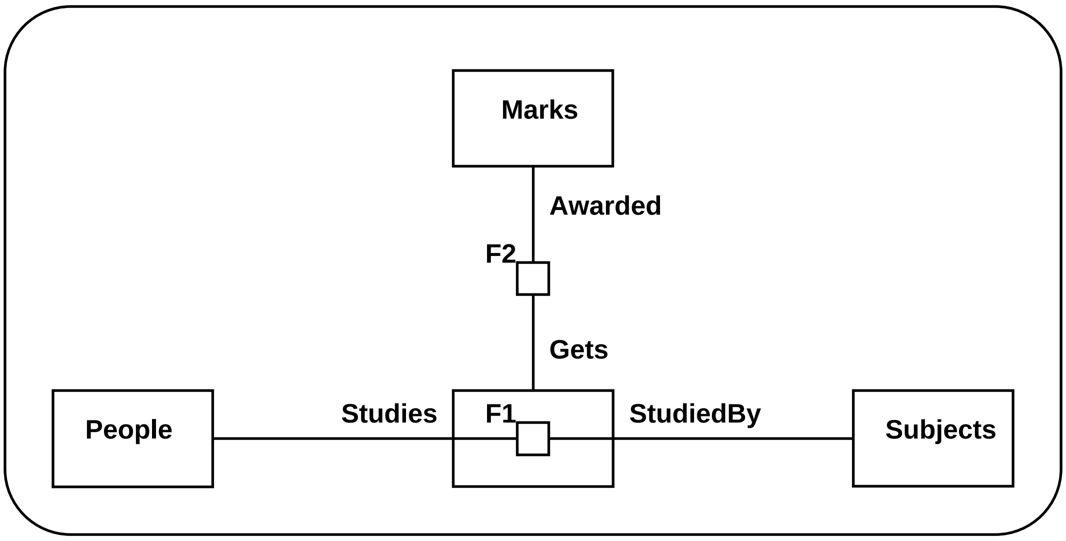
We will build a model that is based on the step by step construction of data models. We will build it in two stages. We will start by introducing a preliminary set PreMod. Each member of PreMod will model a possible data model, but many members will be ill-formed. For instance, some represent data models where lines go nowhere. Then we will define a subset, DaMod0, of PreMod. Each member of DaMod0 is deemed to model a well-formed data model, and later on this will be proved to be so.

Each data model uses certain roles and certain entities. To make it easier to describe the evolution of data models we will say that the roles are drawn from a pool of roles used by all data modellers; likewise for entities. We introduce the non-empty set Roles to model this pool of roles. Similarly, we introduce the non-empty set Entities to model the pool of entities.

Each Entity Type introduced into a data model is defined to be a fixed class of entities. Therefore we model each Entity Type as a subset of Entities. Each tuple defined by a data model is declared to have an index set that is a set of roles. Therefore we model each index set as a subset of Roles.

Now we have the task of modelling Fact Types. Each Fact Type is defined to be a set of tuples forming a cartesian product, so, of course, it is modelled by a cartesian product. The tuples of these cartesian products have indexes that are members of Roles, and elements that are ultimately constructed from members of Entities and Roles. Data modellers do not write a list of tuples when introducing a Fact Type into a data model; they write what they presume is sufficient information to define the cartesian product. Specifically, they introduce an index set and declare that each index, alias role, is associated with a certain domain. From this the cartesian product can be derived if all is well. For instance, in the ad hoc model of our example data model the cartesian product F2 is defined by introducing the index set {Gets, Awarded} and the mapping (Gets $\mapsto$ F1, Awarded $\mapsto$ Marks) from roles to domains. The relevant data model is reproduced below in Figure 4.2.2.1.

Figure 4.2.2.1 One of our example conceptual data models



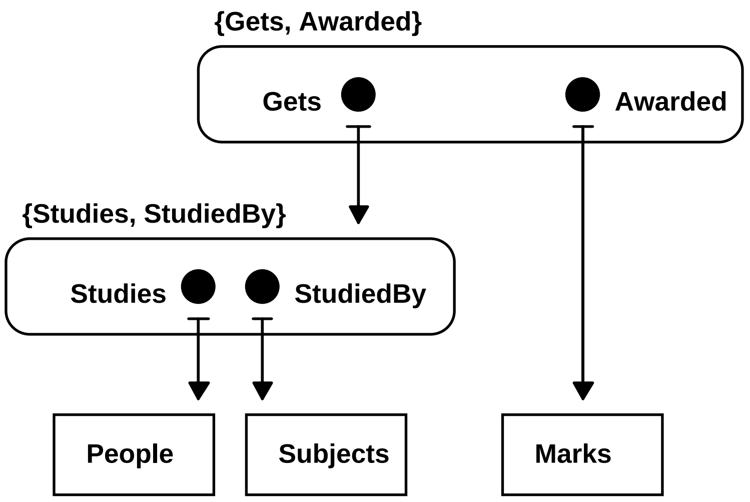
But there is a snag here. It may be that the data model is ill-formed and that the domain F1 is not well defined or does not exist. For instance, if the Entity Type Subjects is removed from the data model then the cartesian product F1 is not fully defined, and so neither is F2. Or suppose that the data model is altered so that F1 is defined by the mapping (Studies $\mapsto$ People, StudiedBy $\mapsto$ F2). We now have a circular definition for F2. It is not clear whether there is a cartesian product obeying the definition or, if there is, whether there is exactly one.

We conclude that we should not regard Fact Types as primary features in our model of data models. What can we do instead? Just as in a database that cannot hold pictures we ask it to hold identifiers instead (names or numbers), so we can use an identifier to indicate which cartesian product a role is supposedly associated with. As can be seen in Figure 4.2.2.1 the domain associated with the role Gets is the Fact Type whose indexes are Studies and StudiedBy. Thus in the model we can use the index set {Studies, StudiedBy} to identify the cartesian product F1. We can define F2 by introducing the mapping (Gets $\mapsto$ {Studies, StudiedBy}, Awarded $\mapsto$ Marks). As this data model is well-formed the index set {Studies, StudiedBy} has the well defined cartesian product F1 associated with it and so we can derive the mapping (Gets $\mapsto$ F1, Awarded $\mapsto$ Marks), and from this we can derive the cartesian product F2.

If the data model is ill-formed then perhaps {Studies, StudiedBy} has no associated cartesian product, but we have still modelled the data model as drawn. We would be able to use the model to explain to a student why the data model is ill-formed rather than just rejecting it out of hand.

The essence of the model of core data models can be illustrated by transforming the data model in Figure 4.2.2.1 into a different notation that conveys exactly the same information : Figure 4.2.2.2 below. There, the diagram introduces some Entity Types and some index sets. Each index set is a set of roles which acts in particular as the identifier for a Fact Type (if such exists). Maplet arrows associate each role in the diagram either with an Entity Type or with an index set, alias identifier.

Figure 4.2.2.2 A different notation



A simple way to model the maplets in Figure 4.2.2.2 is to package them in a single mapping holding all the role associations, giving

(Gets $\mapsto$ {Studies, StudiedBy}, Awarded $\mapsto$ Marks, Studies $\mapsto$ People, StudiedBy $\mapsto$ Subjects),

but we need to be sure that this does not cause any confusion or loss of information. Remember from the example the rule that different Fact Types use different roles. We can be sure that in a well-formed data model each role used belongs to exactly one index set, and that index sets that need to be distinguished are sure to be different. However, we cannot be sure that each index set is different from each Entity Type. We will avoid confusion in our model by declaring that no role is an entity and that no entity is a role. That is, we declare that the two sets Entities and Roles are disjoint. Now each role used is sure to be associated with exactly one object, and there is no doubt which kind of object that is. Thus combining all the role associations into one mapping causes no confusion in the model of a well-formed core data model, and in an ill-formed model it does not matter.

Figure 4.2.2.2 introduces certain Entity Types and index sets. A simple way to model the declaration of these objects is to hold them in a set, giving

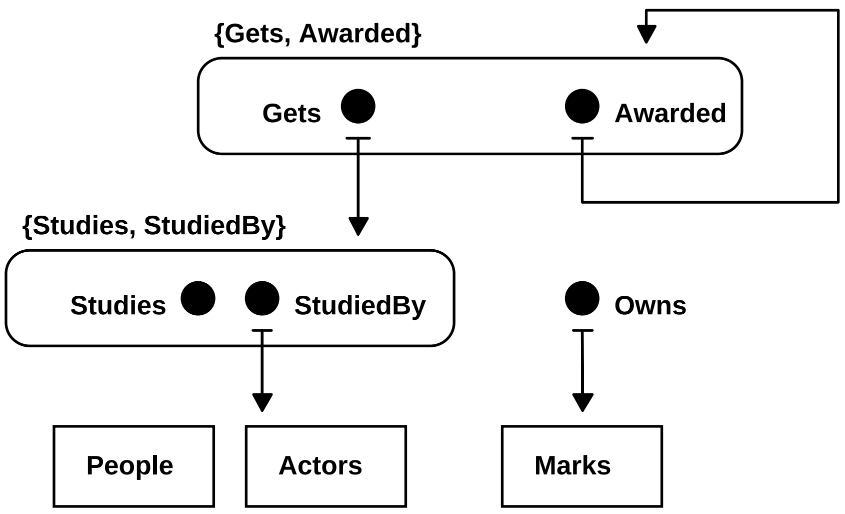
{ {Gets, Awarded}, {Studies, StudiedBy}, Marks, People, Subjects },

but again we need to be sure that this does not cause any confusion or loss of information. As before, we can be sure that in a well-formed data model index sets that need to be distinguished are sure to be different, being non-empty and pairwise disjoint, and in our model we have ensured that any role is different from any entity. Thus in the model of a well-formed data model there is no possibility of confusion or loss of information.

We have modelled the data model of Figure 4.2.2.1 and 4.2.2.2 as an object with two features. The first feature is a set whose members model the Entity Types and index sets introduced by the data model. The second feature is a mapping that models the association of roles with Entity Types and index sets. Together, these two features encapsulate all the information that is relevant to the database and none that is irrelevant, such as the size of the symbols. At this point we need another definition. We define the fixed set Objects to be the set containing all subsets of Entities and all subsets of Roles. Thus each member of Objects models either an Entity Type or an index set (which possibly identifies a Fact Type). Now we can say that the first feature is a subset of Objects, and that the second feature is a mapping from roles to Objects.

We can now generalise this by defining PreMod. Each member *P* of PreMod models a possibly ill-formed core data model and has two primary features. The first feature, *PObjs*, is defined to be a subset of Objects. Each member of *PObjs* models either an Entity Type or an index set that has been introduced by the data model. The second feature, *PConn*, models the association of roles with Entity Types and index sets. To make it easier to describe the evolution of data models, *PConn* is defined to be a partial function from Roles to Objects. That is, it associates some roles with objects. There are no other restrictions on the membership of PreMod. The roles that *PConn* associates with objects need not be relevant, and the objects they are associated with need not be relevant. Obviously, many members of PreMod are ill-formed : Figure 4.2.2.3 gives an example. There, the role Studies lacks an associated object and the associations of Owns with Marks and StudiedBy with Actors are irrelevant. Also, we will see in Section 4.3 that the association of Awarded with {Gets, Awarded} is deemed to be improper.

Figure 4.2.2.3 An ill-formed data model described in PreMod



Having defined PreMod our next task is to define the subset DaMod0 consisting of all those members of PreMod deemed to be well-formed (and proved to be so later on). One way to define DaMod0 would be to state a condition that is true of members of DaMod0 and false of other members of PreMod. Unfortunately, the necessary condition is rather complicated and at first glance it is not obvious that it is both necessary and sufficient.

An alternative way to define DaMod0 is based on the observation that the empty data model is well-formed, and that the "proper" addition of an Entity Type or a Fact Type to a well-formed data model leads to another well-formed data model. In outline we will define well-formed data models to be those that can be constructed from the empty data model by the "proper" addition of one Entity Type or Fact Type at a time. We need a name for the member of PreMod that models the empty data model, and we need to define "proper" addition.

We introduce EmpDaMod0 : PreMod as the model of the empty data model. It introduces no objects, so EmpDaMod0Objs is empty, and it associates no roles with objects, so the definition domain of EmpDaMod0Conn is empty.

We introduce the binary relation AddedEn on PreMod to define the "proper" addition of a set of entities. For any *P*, *P'* : PreMod, *P* AddedEn *P'* iff

P' introduces all the objects that *P* introduces, and one more object;

that additional object is a non-empty set of entities that is disjoint from any of the sets of entities introduced by *P*; (by construction it is disjoint from any set of roles); and

P and *P'* define the same associations of roles with objects.

Intuitively, if *P* is well-formed then *P'* is well-formed. That is, for each index set, alias set of roles, there is a well defined cartesian product. (The intuitions will be proved later on).

We introduce the binary relation AddedRo on PreMod to define the "proper" addition of an index set, alias set of roles. For any *P*, *P'* : PreMod, *P* AddedRo *P'* iff

P' introduces all the objects that *P* introduces, and one more object;

that additional object is a non-empty, finite, set of roles that is disjoint from any of the sets of roles introduced by *P*; (by construction it is disjoint from any set of entities);

each role of the index set is associated by *P'* with an object introduced by *P*;

and for other roles, *P* and *P'* define the same associations of roles with objects.

Intuitively, if *P* is well-formed then *P'* is well-formed : the cartesian products defined by *P* are unchanged and there is one additional cartesian product associated with the added index set which is clearly well defined.

Finally, DaMod0 is defined to be the smallest of the subsets *A* of PreMod that have the two properties :

EmpDaMod0 is a member of *A*; and

If *P* is a member of *A*, *P'* is a member of PreMod, and *P* AddedEn *P'* or *P* AddedRo *P'*, then *P'* is also a member of *A*.

That is, DaMod0 is the inductively generated subset of PreMod whose generators are {EmpDaMod0, AddedEn, AddedRo}.

We can use this definition to show that the first four steps of our example data model are modelled by members of DaMod0 and so are deemed to be well-formed. At each step we define a member D_1, D_2, D_3, or D_4 of PreMod and then show that it must be a member of DaMod0 :

Step 1 Initial, empty, core model $D_1 =_d$ EmpDaMod0

D_1Objs = { }
D_1Conn = ( ), the empty mapping
D_1 = EmpDaMod0 so $D_1 \in$ DaMod0

Step 2 Add People, a non-empty set of entities, to D_1 giving D_2 : PreMod

D_2Objs = { People }
D_2Conn = ( ), the empty mapping
$D_1 \in$ DaMod0 and D_1 AddedEn D_2 so $D_2 \in$ DaMod0

Step 3 Add Subjects, an entirely different non-empty set of entities, to D_2 giving D_3 : PreMod

D_3Objs = { People, Subjects }
D_3Conn = ( ), the empty mapping
$D_2 \in$ DaMod0 and D_2 AddedEn D_3 so $D_3 \in$ DaMod0

Step 4 Add the index set {Studies, StudiedBy}, a set of two different roles, to D_3 giving D_4 : PreMod
Associate Studies with People and StudiedBy with Subjects

D_4Objs = { People, Subjects, {Studies, StudiedBy} }
D_4Conn = (Studies $\mapsto$ People, StudiedBy $\mapsto$ Subjects)
$D_3 \in$ DaMod0 and D_3 AddedRo D_4 so $D_4 \in$ DaMod0

D_4 models a core data model defining one Fact Type whose informal description is 'People study Subjects'.

This completes the outline definition of the model of core data models. Observe that we have a system with three Levels of Variation (see Section 3.3.3) :

Level 1 : A description of all core data models.

Level 2 : A single core data model, which if it is well-formed defines the set of all interesting database instances (some legitimate, some not).

Level 3 : A single database instance (a particular set of tuples).

Operations within the different levels are quite distinct. Operations in Level 1 transform one data model into another or compare data models with each other. They model the evolution of data models during the database design process and any subsequent changes resulting from changed requirements. These operation are the subject of Chapter 5.

Operations in Level 2 transform one database instance into another. They model the evolution of database contents resulting from user actions. These operations are outside the scope of this work.

Operations in Level 3 act on a single database instance without changing it; they can only report the instance's properties. They model database queries by users. These operations are also outside the scope of this work.

There is also a somewhat vaguely defined Level 0 in which we vary the model of data models. Operations in this level establish equivalences or essential differences between different models. We do not define such operators in this work.

4.2.3 Definition details

This section provides the detailed definition of the set-theoretical model of core data models. The definition uses Scheurer's Feature Notation (outlined in Section 3.3).

First we introduce the two primitive sets Entities and Roles, and the set Objects derived from them.

Entities : Set

The set modelling all possible entities that might be used by data modellers.

$\text{Entities} \neq \emptyset$

Roles : Set

The set modelling all possible roles, alias indexes, that might be used by data modellers.

$\text{Roles} \neq \emptyset$

$\text{Roles} \cap \text{Entities} = \emptyset$ No role is an entity.
Any non-empty set of roles is distinct from any non-empty set of entities

---

The sets Entities and Roles are defined to be non-empty but are not further determined.

Objects : Set

All possible sets of entities, each modelling an Entity Type, and all possible sets of roles, each modelling an index set (and in a well-formed model acting as the identifier for a cartesian product modelling a Fact Type).

$\text{Objects} =_d \text{Pow(Entities)} \cup \text{Pow(Roles)}$

Note that the object ∅ is potentially ambiguous but as well-formed data models do not use the empty Entity Type or the empty index set this does not matter.

Next we define the set PreMod, whose members model all core data models deemed to be well-formed and many that are ill-formed.

$\underline{P : \text{PreMod}}$

Precursor class of all models of possibly ill-formed core data models.

$*$ $PObjs \subseteq_d$ Objects	The set of objects introduced by this core model
$*$ $PConn$: Roles $+\!\!\rightarrow$ Objects	Partial function associating some possibly irrelevant roles with possibly irrelevant objects

---

Note that PreMod is a set, not a proper class.

Finally we define the set DaMod0 $\subseteq_d$ PreMod whose members model those core data models deemed to be well-formed (and later on proved to be so). DaMod0 is an inductively generated subset of PreMod.

We start the definition of DaMod0 by defining its three generators : EmpDaMod0, AddedEn, and AddedRo.

EmpDaMod0

Nullary function, alias constant, that returns the (unique) empty core model

EmpDaMod0 $=_d P$ where P : PreMod and

$PObjs =_d \varnothing$	No objects are introduced
$PConnDef =_d \varnothing$	No roles are associated with objects

---

AddedEn

Predicate that is true of P, P' : PreMod iff the change from P to P' is the addition of one suitable set of entities, with no change of role associations.

Given any P, P' : PreMod then

P AddedEn P' $\Leftrightarrow_d$

$\exists\, E \subseteq_d$ Entities $\bullet$	There is a set of entities,
$E \neq \varnothing \;\wedge$	that is not empty,
$[\, \forall x : PObjs \;\bullet\; x \cap E = \varnothing \,] \;\wedge$	that is not a member of *PObjs*, and is disjoint from each member of *PObjs*,
$P'Objs = PObjs \cup \{E\} \;\wedge$	that when added to P gives P',
$P'Conn = PConn$	with no change of connections

AddedRo

Predicate that is true of P, P' : PreMod iff the change from P to P' is the addition of one suitable index set, alias set of roles, with additional suitable role associations.

Given any P, P' : PreMod then

P AddedRo P' $\Leftrightarrow_d$

$\exists\ R \subseteq_d \text{Roles} \bullet$	There is a set of roles,
$R \neq \varnothing \ \wedge\ \text{IsFinite}(R)\ \wedge$	that is non-empty and finite,
$[\ \forall x : PObjs \ \bullet\ x \cap R = \varnothing\ ]\ \wedge$	that is not a member of *PObjs*, and is disjoint from each member of *PObjs*,
$P'Objs = PObjs \cup \{R\}\ \wedge$	that when added to P gives P',
$[\ \forall r : R \ \bullet\ r \in P'ConnDef\ \wedge\ P'Conn(r) \in PObjs\ ]\ \wedge$	with each role associated with an object of P,
$(\text{Roles} - R) \langle\ P'Conn = (\text{Roles} - R) \langle\ PConn$	with no other changes

It is convenient to give the name Gen to the set of generators.

Gen

The set of generators used to generate DaMod0

Gen $=_d$ { EmpDaMod0, AddedEn, AddedRo }

Next we define what it means to say that a subset of PreMod is inductive with respect to Gen.

IsInductive

Predicate that is true of $A \subseteq_d$ PreMod iff A is inductive with respect to the set Gen of generators

Given any $A \subseteq_d$ PreMod then

IsInductive(A) $\Leftrightarrow_d$
  EmpDaMod0 $\in A\ \wedge$
  $\forall\ P, P'$: PreMod $\bullet$
   $[\ (\ P \in A\ \wedge\ (\ P$ AddedEn $P'\ \vee\ P$ AddedRo $P'\ )\ ) \Rightarrow P' \in A\ ]$

And finally we define DaMod0 to be the smallest inductive subset of PreMod.

DaMod0 $\subseteq_d$ PreMod

The set of all well-formed, but not necessarily complete, models of core data models. DaMod0 is the subset of PreMod inductively generated by Gen.

$$\text{DaMod0} =_d \bigcap \{ A \subseteq_d \text{PreMod} \mid \text{IsInductive}(A) \}$$

DaMod0 is not freely generated : most data models can be built incrementally in more than one order. For instance, if there are two Entity Types then either could be added first. We will see in the next section that this is not a disadvantage.

Note that DaMod0 is uniquely determined once the sets Roles and Entities have been fixed. We could introduce a higher Level of Variation by varying the membership of Roles and Entities. However, there does not appear to be anything to be gained by doing so.

This completes the definition of the model of core data models.

4.3 The cartesian products determined by any *D* : DaMod0

The purpose of this section is to prove that each member of DaMod0 is well-formed. Obviously, we must define "well-formed" in a way that matches the requirements of data modellers. During the proof we will introduce many definitions, properties, and proof techniques that will be re-used in later sections and chapters.

As before, the text is presented as an outline followed by mathematical details.

4.3.1 Outline

Recall from the previous section that each member *P* of PreMod has two primary features : the set *PObjs* of objects; and the partial function *PConn* that associates some roles with objects. Each object is either a set of entities or a set of roles, alias index set. The criterion for *P* being well-formed is that each index set *R* belonging to *PObjs* identifies an appropriate and well defined cartesian product. To be well defined our construction must assign exactly one cartesian product to *R*. This will be so only if each role of *R* is associated by *PConn* with an object that identifies a well defined domain. The domain can be identified directly as a set of entities or indirectly through an index set identifying a cartesian product. Our task is to prove that every member of DaMod0, the specially selected subset of PreMod, meets this criterion.

As it happens, we will not prove that there is a well defined cartesian product for each index set. Instead, we will prove the more general property that a certain kind of function definition is valid. It is then straightforward to define the function that we need here. To prove the general property we need to prove that the internal structure of each member of DaMod0 has a certain property. To do this, we must introduce a proof technique that can be used to prove the property. To help us with all this we need to introduce some terms for the component parts of each member of PreMod, and some more terms that will be applied only to members of DaMod0. Detailed definitions and proofs are given in Section 4.3.2; we give only outlines in this section.

Recall from Section 3.3 that a secondary feature is a feature that is derived from primary features. Any secondary feature is fixed once the primary features are fixed. We will define some secondary features of the members of PreMod. Suppose we have a member of PreMod that introduces the set

{ People, Subjects, {Studies, StudiedBy} }

of objects. We would like to have names for the subset {People, Subjects} consisting of sets modelling Entity Types, for the subset { {Studies, StudiedBy} } consisting of sets modelling index sets, for the set {Carol, Jim, Physics, …} of all entities used, and for the set {Studies, StudiedBy} of all roles used.

For any *P* : PreMod we define the following secondary features. Each defines a use of the phrase "occurring in".

PEnSets : The members of *PObjs* that are subsets of Entities. I.e The objects occurring in *P* that model Entity Types.

PRoSets : The members of *PObjs* that are subsets of Roles. I.e The objects occurring in *P* that model index sets.

PEntities : The union of the members of *PEnSets*. I.e The entities belonging to some member of *PObjs*.

PRoles : The union of the members of *PRoSets*. I.e The roles belonging to some member of *PObjs*.

Next we turn to a property of DaMod0 as a whole. DaMod0 is an inductively generated set so we can use the proof technique called structural induction to prove properties of DaMod0. (See Scheurer [1994], p213-225, for a thorough treatment of structural induction). The general form of such a proof goes as follows. Suppose we wish to prove that some property Prop is true of all members of DaMod0. We prove that

1) Prop is true of EmpDaMod0, the empty model; and
2) for any members *D*, *D′* of DaMod0, whenever Prop is true of *D* and also *D* AddedEn *D′* then Prop is true of *D′*; and
3) for any members *D*, *D′* of DaMod0, whenever Prop is true of *D* and also *D* AddedRo *D′* then Prop is true of *D′*.

This is all that is needed to prove that Prop is true of all members of DaMod0. In effect, we prove that whenever we build a data model by starting with the empty data model and adding one Entity Type or Fact Type at a time in a proper manner then Prop is true at each step; hence it is true for any data model that we build this way.

We will illustrate the technique by proving several simple properties. Together they show that each core data model described by a member of DaMod0 is tidy in the ways that we would expect. Specifically, we will prove for any *D* : DaMod0 that

a) *DObjs* is finite. *D* describes a data model whose diagram contains a finite number of Entity Type and Fact Type symbols.
b) *DObjs* is a set of non-empty pairwise disjoint sets. *D* describes a data model with non-empty Entity Types that are pairwise disjoint and non-empty index sets that are pairwise disjoint.
c) *DConnDef* = *DRoles*. (*DConnDef* is the definition domain of the function *DConn*). *D* describes a data model in whose diagram there is a line for each role of each Fact Type symbol, and for no other roles.
d) *DConnRan* $\subseteq$ *DObjs*. (*DConnRan* is the range of the function *DConn*). *D* describes a data model in whose diagram every line joins two symbols occurring in the diagram.
e) *DRoles* is finite. *D* describes a data model whose diagram contains a finite number of lines.

First, take the case of EmpDaMod0, which introduces no objects and associates no roles with objects. Zero is a finite number so (a) is true. (b) and (d) are vacuously true. No roles occur in EmpDaMod0 so (e) is true, and no roles are associated with objects, so (c) is also true. Thus properties (a) to (e) are true of EmpDaMod0.

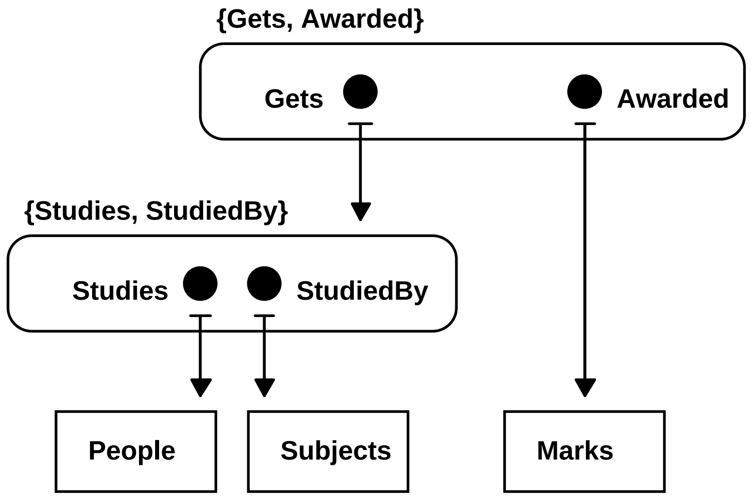
Second, take any *D*, *D′* : DaMod0 with *D* AddedEn *D′*. From the definition of AddedEn (Section 4.2.3) we know that *D′* introduces exactly one more object than *D*. Thus if (a) is true of *D* then it is true of *D′*. Moreover, that object is a non-empty set that is disjoint from each object of *D*. Thus if (b) is true of *D* then it is true of *D′*. Also, that object is a subset of Entities. The roles occurring in *D′* are the same as the roles occurring in *D*, and *D′* has the same association of roles with objects as *D* does. Thus if (c), (d), and (e) are true of *D* then they are equally true of *D′*. In total, if properties (a) to (e) are true of *D* then they are true of *D′*.

Third, take any *D*, *D′* : DaMod0 with *D* AddedRo *D′*. From the definition of AddedRo (Section 4.2.3) we know that *D′* introduces exactly one more object than *D* and that that object is a non-empty set, disjoint from each object introduced by *D*. As in the second case, if (a) and (b) are true of *D* then they are true of *D′*. The additional object is a finite set of roles. Thus if (e) is true of *D* then it is true of *D′*. Also, from the definition of AddedRo we know that each role of the added object is associated with an object already introduced by *D* and that other role associations are unchanged. Thus if (c) and (d) are true of *D* then they are true of *D′*. Again, in total, if properties (a) to (e) are true of *D* then they are true of *D′*.

From these three cases we conclude that properties (a) to (e) are true of every member of DaMod0. Therefore each member of DaMod0 models a core data model that can be drawn with a finite number of Entity Type and Fact Type symbols and a finite number of lines. The diagram for such a data model may be too large to be drawn in practice but doing so is not inherently impossible. Moreover, each line joins two symbols of the diagram, as it should do.

From now on we will concentrate on individual members of PreMod and DaMod0. We will start by defining two more secondary features, defined for each member of PreMod. Consider Figure 4.3.1.1 which describes a member of PreMod using the ad hoc notation from the previous section. We will define two relations that give a partial description of the member's structure.

Figure 4.3.1.1 A member of PreMod



We will say that the object {Gets, Awarded} **uses** the object {Studies, StudiedBy} as there is a maplet arrow from a role of {Gets, Awarded} to {Studies, StudiedBy}. Similarly, we will say that {Studies, StudiedBy} uses the object Subjects but that Subjects does not use anything. Conversely, we will say that Subjects is **used by** {Studies, StudiedBy} and that {Studies, StudiedBy} is used by {Gets, Awarded}.

For any *P* : PreMod we define two relations, *PUses* and its opposite *PUsedBy*, to express this usage :

PUses :	For any objects *x* and *y* that are members of *PObjs*, *x PUses y* iff *x* contains a role that is mapped to *y* by the function *PConn*.
PUsedBy :	For any objects *x* and *y* that are members of *PObjs*, *y PUsedBy x* iff *x* contains a role that is mapped to *y* by the function *PConn*. Equivalently, *PUsedBy* $=_d$ Opp(*PUses*).

Notice that there may be several maplet arrows from one object to another but this is hidden by *PUses* and *PUsedBy*. These relations provide only partial information about *P*. If *P* is a member of DaMod0 then *PUses* and *PUsedBy* have a key property, but before discussing it we will introduce two more secondary features. They are defined only for members of DaMod0 as their names could be inappropriate in some other members of PreMod.

Referring to Figure 4.3.1.1 again, we will say that both the objects {Studies, StudiedBy} and Marks are **immediate predecessors** of {Gets, Awarded}, and that People and Subjects are immediate predecessors of {Studies, StudiedBy}. They are predecessors in the sense that a well defined cartesian product must be given to {Studies, StudiedBy} before one can be given to {Gets, Awarded}. The objects People, Subjects, and Marks have no immediate predecessors. Conversely, we will say that {Studies, StudiedBy} is an **immediate successor** of Subjects, (there is only one successor here as it happens), and that {Gets, Awarded} is an immediate successor of {Studies, StudiedBy}.

For any *D* : DaMod0 we define two functions, *DPreds* and *DSuccs*, to express this usage :

DPreds :	Given any object *t* : *DObjs* then *DPreds*(*t*) is the set of immediate predecessors of *t*; specifically those members *x* of *DObjs* for which *x DUsedBy t*.
DSuccs :	Given any object *t* : *DObjs* then *DSuccs*(*t*) is the set of immediate successors of *t*; specifically those members *x* of *DObjs* for which *t DUsedBy x*; (equivalently, for which *x DUses t*).

Now we have the terminology needed to discuss well defined values. Suppose we pick some Fact Type F in a data model and wish to find the Entity Types that F ultimately depends on. We inspect each of F's immediate predecessors. If it is an Entity Type then it is one of the Entity Types that F depends on. If it is a Fact Type, F' say, then F depends on each of the Entity Types that F' depends on. For instance, in Figure 4.3.1.1 we can see that {Gets, Awarded} depends on Marks, and on anything that {Studies, StudiedBy} depends on, which is People and Subjects. If all is well we have a well defined set of Entity Types that F depends on. However, to find what F' depends on we must look at each of the immediate predecessors of F', and at their immediate predecessors, if any, and so on. If we eventually arrive at an Entity Type every time then all is well, but if we find we have gone round in a circle, for instance if we reach F' again, then we do not know what to say.

A similar situation arises in simple equations. Finding the values of *x* and *y* when you are given

$y = 7 + x$ and $x = 2 + 3$
is straightforward, but finding x when you are given
$x = 2 - x$ or $x = 2 + x$
is not so straightforward and might not be possible, as in the second equation.

We would prefer the straightforward case, and in defining DaMod0 we have ensured that we have the straightforward case in every member of DaMod0. The process of following immediate predecessors, their immediate predecessors, and so on, never goes round in a circle. We always arrive at sets of entities, which have no immediate predecessors. To use the proper words, we will prove in the details section that for any *D* : DaMod0, *DUsedBy* is a Well Founded relation. This property of *D* gives us another proof technique and a technique for introducing well defined functions on *DObjs*. (See Enderton [1977], p241-249, for a thorough treatment of Well Founded relations and proofs of the validity of the two techniques).

The proof technique will be called Well Founded induction to distinguish it from the structural induction technique described earlier. The general form of such a proof goes as follows. Suppose we have some arbitrary member *D* of DaMod0 and we wish to prove that some property Prop is true of all the members of *DObjs*. We prove for each member *t* of *DObjs* that

If Prop is true of every immediate predecessor of *t*, then it is also true of *t*.

This is all that is needed to prove that Prop is true of all the members of *DObjs*. Note that if *t* has no immediate predecessors, that is, when *t* is a set of entities, then Prop must be proved true unconditionally. In effect, we prove that the property is true of *t* if it is true of the objects that *t* ultimately depends on, and we also prove that it is true of them. As *D* is an arbitrary member of DaMod0 we can now conclude that Prop is true of the objects of any member of DaMod0.

The function definition technique will be called Well Founded recursion to distinguish it from other kinds of recursion. The general pattern for a definition is as follows. Suppose we have some arbitrary member *D* of DaMod0 and we wish to define a function *f* that is to be defined on *DObjs*. We need only state that

For any member *t* of *DObjs*, *f*(*t*) is a given function G of
t, and of
the immediate predecessors *t'* of *t*, and of
their values *f*(*t'*).

The function *f* is sure to be well defined. That is, such a function exists and is unique. (More accurately, its graph is unique. There will often be a choice of codomain.) To put it another way, we define *f*(*t*) in such a way that if each of *t*'s immediate predecessors has a well defined value then so does *t*. Note that when *t* is a set of entities, and so has no immediate predecessors, then *f*(*t*) is a function of *t* or is constant. As *D* is an arbitrary member of DaMod0 we can conclude that there is a well defined function, defined as above, on the objects of any member of DaMod0. The function is a secondary feature. Different functions G give us different secondary features.

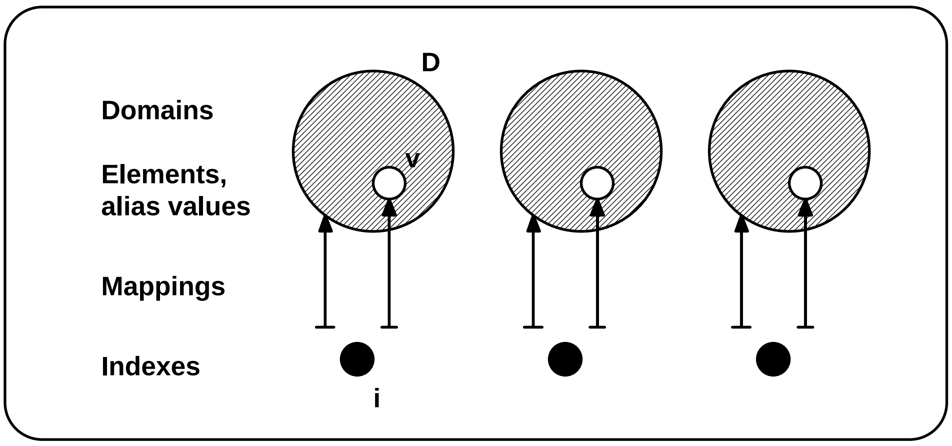
Now we can discuss cartesian products. For each *D* : DaMod0 we wish to assign an appropriate and well defined cartesian product to each index set belonging to *DObjs*. If *t* : *DObjs* is an index set, such as {Gets, Awarded} in Figure 4.3.1.1, then we have a well defined cartesian product if each role of *t* is associated with a well defined domain. Suppose we have the role *r* of *t* and *DConn*(*r*) = *x*. Then *x* is a member of *DObjs* and is an immediate predecessor of *t*. If *x* is a set of entities then the domain associated with *r* is *x* itself, which is well defined. Thus the role Awarded has the domain Marks. If *x* is an index set then we hope that there is a well defined cartesian product associated with *x*. If so, then this cartesian product is the domain associated with *r*. Thus the role Gets has the cartesian product assigned to {Studies, StudiedBy} as its domain. The cartesian product to be assigned to *t* is a function of *t* (index set), of *t*'s immediate predecessors, and their cartesian products if appropriate (domains). But this definition fits the pattern of Well Founded recursion. Consequently we know that there is a well defined function, which we will call *DCart*, that assigns a cartesian product defined in this way to each index set of *DObjs* (and some default value to any other members of *DObjs*).

This result applies to any member of DaMod0. We can now be sure, given some proofs in the details section, that any core data model described by a member of DaMod0 specifies well defined Fact Types and so is well-formed, which is what we set out to prove.

We will continue a little further. Proofs and definitions appearing later on are made simpler if we use an explicit representation of tuples. We then need to define an operator that constructs cartesian products consisting of this kind of tuple. The detailed definition of *DCart* will use this operator.

Tuples that are represented this way will be called **fact-style tuples** for want of a better name. A minimum requirement is that each tuple contains a mapping from an index set to values, alias elements. Such a mapping will be called a **value function**. In NIAM, each element of a tuple is constrained to belong to a fixed domain. A mapping from an index set to domains will be called a **domain function**. A somewhat arbitrary modelling decision has been taken to represent a fact-style tuple as an object containing an index set, a domain function, and a value function. Figure 4.3.1.2 illustrates this.

Figure 4.3.1.2 Representation of a fact-style tuple



In more detail, we declare that each member *T* of the class Tuple of all fact-style tuples has these three features :

TI : Index set, which can be any set;

TDomf : Domain function, mapping each index to a domain which can be any non-empty set;

TVal : Value function, mapping each index to a value which can be any member of the index's domain.

Thus a fact-style tuple is a distributed function (as in Scheurer [1994], p136-137) or an "I-tuple" (as in Enderton [1977], p54) represented in a very specific way. The inclusion of the domain function inside each tuple is rather like writing 'Person Carol studies Subject Physics' instead of the simpler 'Carol studies Physics'. Note that Tuple has been defined to be general. In the tuples defined by members of DaMod0 each index set is a finite set of roles.

As usual, the next step after defining tuples is to define cartesian products. A **fact-style cartesian product** is a set containing all the members of Tuple that have a particular domain function (and so have the same index set). The operator that we will call CartProd returns the appropriate fact-style cartesian product when given a domain function. Unlike the usual operator Π, alias **X**, it returns a set of fact-style tuples.

There is exactly one member of Tuple whose index set has no members : the nullary tuple, which we will call ϕ (not to be confused with $\varnothing$). As its index set is empty it maps no indexes to domains and no indexes to values. If CartProd is given the empty domain function then it returns the unique nullary fact-style cartesian product, which we will call Φ. It contains the single nullary tuple ϕ. The nullary tuple is not allowed in database instances (Section 3.1.3, point 20) but ϕ and Φ can appear in definitions or in the result returned by a database query that is certain to be fruitless.

Now we can return to DaMod0. For any *D* : DaMod0 and object *t* : *DObjs* the cartesian product *DCart*(*t*) is defined to be a fact-style cartesian product. It is formed by applying the operator CartProd to the appropriate domain function. For instance, in Figure 4.3.1.1 the cartesian product assigned to {Studies, StudiedBy} is

$\text{C} =_d \text{CartProd}((\text{Studies} \mapsto \text{People}, \text{StudiedBy} \mapsto \text{Subjects}))$,

and to {Gets, Awarded} it is

$\text{CartProd}((\text{Gets} \mapsto \text{C}, \text{Awarded} \mapsto \text{Marks}))$.

If *t* is not an index set, i.e if it is a set of entities, then it is assigned the default value CartProd($\varnothing$), alias Φ.

If we form the union of the cartesian products *DCart*(*t*) for each index set *t* then we have the set *DFacts* of all tuples defined by *D*. Each subset of *DFacts* models an instance of the database described in part by *D*. If we now form the union of *DFacts* over DaMod0 we have the set Facts of every fact-style tuple defined by some member of DaMod0. Note that Facts is uniquely determined by the sets Entities and Roles.

4.3.2 Details

This section provides definitions and properties needed to show that each member of DaMod0 is well-formed. Proofs are included when appropriate.

Properties are stated in Feature Notation form. Each property is a secondary fixed feature with a feature name such as *DProp1.5*. The naming scheme may appear erratic. This is because they are maintained in an unpublished list of properties found useful when designing proofs. Most of the properties are straightforward or trivial and are not mentioned in this work.

Proofs start with the words "**To prove**" in bold and end with the Halmos symbol "□".

We start with two groups of secondary features defined on all members of PreMod. The first group gives names to component parts.

P : PreMod Secondary Features 1

Some secondary features of each member of PreMod : Components

$PEnSets =_d PObjs \cap \mathrm{Pow(Entities)}$	The objects occurring in *P* that model Entity Types
$PRoSets =_d PObjs \cap \mathrm{Pow(Roles)}$	The objects occurring in *P* that model index sets, alias Fact Type identifiers
$PEntities =_d \bigcup PEnSets$	The entities occurring in *P*
$PRoles =_d \bigcup PRoSets$	The roles occurring in *P*

The second group gives an incomplete but important description of a data model's structure. Here and elsewhere, when we use formulas to define relations and functions the definitions are stated in Feature Notation form. Each definition is a secondary fixed feature with a feature name such as *PDefUses*.

P : PreMod Secondary Features 2

Some secondary features of each member of PreMod : Structure

$PUses : PObjs \leftrightarrow PObjs$	For any *x*, *y* : *PObjs*, *x PUses y* iff some role in *x* is associated with *y*

$PDefUses$.: $\forall x, y : PObjs \bullet$
$x\ PUses\ y \Leftrightarrow_d \exists r : PConnDef \bullet r \in x \wedge PConn(r) = y$

$PUsedBy : PObjs \leftrightarrow PObjs$	For any *x*, *y* : *PObjs*, *y PUsedBy x* iff *y* is associated with some role in *x*

$PDefUsedBy$.: $PUsedBy =_d \mathrm{Opp}(PUses)$

Now we turn to the properties of DaMod0 as a whole. DaMod0 is an inductively generated subset of PreMod. We should show that DaMod0 has been properly defined. Clearly, PreMod exists, PreMod is not empty, and EmpDaMod0 is a member of PreMod. Thus DaMod0 is defined in a proper manner as the intersection of a non-empty set; DaMod0 exists and is uniquely determined. We can construct three members of DaMod0 by using one entity and one role so we have a model that is neither vacuous nor trivial.

The properties of inductively generated sets, with proofs, are treated at length in Scheurer [1994], p213-225 and briefly in Enderton [1972], p22-25. We will apply their results here : DaMod0 has an induction principle and a structural induction proof schema derived from it.

In induction proofs we define the desired property at a point t by a Wff α in which the variable t is free, e.g $t < 3$. We then define the property at any point x by the Wff α^t_x where each free occurrence of t has been replaced by x, e.g $x < 3$. For this to be proper, x must be free in α^t_x wherever it replaces t. That is, x must be substitutable for t in α. A detailed definition of "substitutable" is given in Enderton [1972], p104-106. Where Feature Notation is used replacement is extended to features. For instance, if *D* is replaced by *D'* then *DRoles* is also replaced by *D'Roles*.

DaMod0 Properties 1

Some properties of the set DaMod0 : Structural induction

DaMod0Prop3 .: Induction principle for DaMod0

$\forall A \subseteq_d \text{DaMod0} \bullet \text{IsInductive}(A) \Rightarrow A = \text{DaMod0}$

Note : DaMod0 is **not** freely generated

DaMod0Prop4 .: Structural induction proof schema for DaMod0

Given any Wff α such that *D'* is substitutable for *D* in α,

Let $\alpha' =_d \alpha^{D}_{D'}$, then

$[\ \forall D, D' : \text{DaMod0} \bullet$
$(\ D = \text{EmpDaMod0} \Rightarrow \alpha\ ) \wedge$
$(\ D \text{ AddedEn } D' \Rightarrow (\ \alpha \Rightarrow \alpha'\ )\ ) \wedge$
$(\ D \text{ AddedRo } D' \Rightarrow (\ \alpha \Rightarrow \alpha'\ )\ )\ ]$
$\Rightarrow$
$\forall D : \text{DaMod0} \bullet \alpha$

From now on we will concentrate on the properties of each individual member of DaMod0. Some properties were proved in the outline section. They are repeated here to give them names.

D : DaMod0   Properties 1

Some properties of each member of DaMod0 : Tidiness

DProp1.1 .: IsFinite(*DObjs*)

DProp1.2 .: IsFinite(*DRoles*)

DProp1.3 .: *DConnDef* = *DRoles*

DProp1.5 .: *DConnRan* $\subseteq$ *DObjs*

DProp1.7 .: $\forall x$: *DObjs* $\bullet$ $x \neq \varnothing$

DProp1.8 .: *DObjs* is pairwise disjoint
$\forall x, y$: *DObjs* $\bullet$ $x = y \vee x \cap y = \varnothing$

Before going on to the main business we introduce four features of any *D* : DaMod0. The first is occasionally useful when stating properties : if *r* : *DRoles* then there is exactly one object of *DObjs* containing *r*; *DRo*(*r*) is that object. Each object *t* : *DObjs* will be associated with some tuples. *DArity*(*t*) is the arity of those tuples. *DPreds* and *DSuccs* were described in the outline section.

D : DaMod0   Secondary features 1

Some secondary features of each member of DaMod0

DRo : *DRoles* $\rightarrow$ *DObjs*	Given *r* : *DRoles* then *DRo*(*r*) is the (unique) member of *DObjs* containing *r*

DDefRo .: $\forall r$: *DRoles* $\bullet$ $\forall R$: *DObjs* $\bullet$ *DRo*(*r*) = *R* $\Leftrightarrow_d$ $r \in R$

DArity : *DObjs* $\rightarrow$ Nat	Given *t* : *DObjs* then *DArity*(*t*) is the number of roles in *t*. Objects modelling Entity Types have arity 0.

DDefArity .: $\forall t$: *DObjs* $\bullet$ *DArity*(*t*) $=_d$ Num($t \cap$ Roles)

DPreds : *DObjs* $\rightarrow$ Pow(*DObjs*)	Given *t* : *DObjs* then *DPreds*(*t*) is the set of immediate predecessors of *t* w.r.t. *DUsedBy*

DDefPreds .: $\forall t$: *DObjs* $\bullet$ *DPreds*(*t*) $=_d$ { *x* : *DObjs* | *x DUsedBy t* }

DSuccs : *DObjs* $\rightarrow$ Pow(*DObjs*)	Given *t* : *DObjs* then *DSuccs*(*t*) is the set of immediate successors of *t* w.r.t. *DUsedBy*

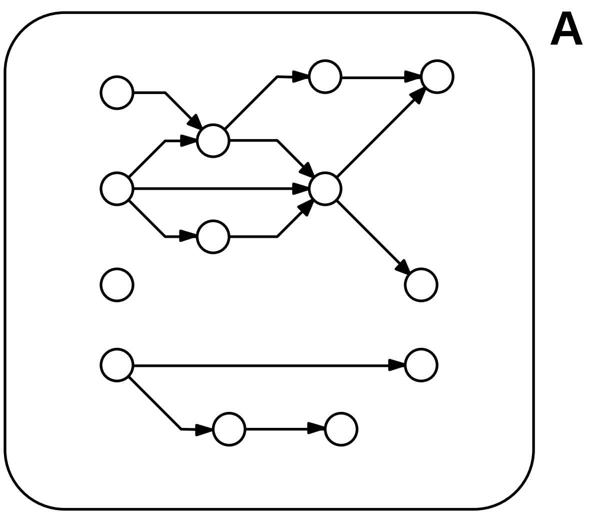
DDefSuccs .: $\forall t$: *DObjs* $\bullet$ *DSuccs*(*t*) $=_d$ { *x* : *DObjs* | *t DUsedBy x* }

We now come to the main result : for any *D* : DaMod0 the relation *DUsedBy* is a Well Founded relation, with useful consequences. We will start by giving a brief introduction to Well Founded relations in general. Then we will present their properties as they apply to the members of DaMod0. This will be followed by the proof that *DUsedBy* is Well Founded, followed by an example of the use of the recursion theorem for Well Founded relations.

A thorough treatment of Well Founded relations, with proofs, is given in Enderton [1977], p241-249. His results are applied here.

The main characteristic of a Well Founded relation is illustrated in Figure 4.3.2.1 below. The circles represent the members of a set *A*. The arrows represent the graph of a relation *R* : *A* ↔ *A*.

Figure 4.3.2.1 Picture of the Well Founded relation R : A ↔ A



Certain members of *A*, shown at the left, have no arrows leading to them. These members will be called base, alias *R*-minimal, members of *A*. It is a characteristic feature of Well Founded relations that, starting at the base members on the left, the arrows lead one inevitably further and further to the right. Every member of *A* can be reached by starting at a base member and following successive arrows. Furthermore, if we start at any member of *A* and follow arrows backwards along any path we will always reach a base member in a finite number of steps. This would be true even if *A* were infinite. Note that it is only a convention that says that the arrows lead away from the base members. The definition could equally have said that the arrows lead towards the base members.

For the general case there are two equivalent definitions of Well Founded relations. One definition uses the reverse path property mentioned above. The other definition is perhaps less obvious but is more convenient for proofs and will be used here.

Suppose now that *A* is any set and *R* : *A* ↔ *A* is any relation on *A*. *R* is a Well Founded relation iff every non-empty subset *X* of *A* contains at least one member *t* with a special property. The property is that there is no member *x* of *X* for which *x R t*. In DaMod0 terminology, none of *t*'s immediate predecessors is a member of *X*.

If *R* is Well Founded then there is an induction principle, a proof schema, and a recursion theorem for *A* (proofs in Enderton [1977]). Returning to Figure 4.3.2.1, the arrows show how properties and definitions flow from the base points to all other points of *A*.

The general properties given in Enderton are translated below into properties of any *D* : DaMod0, subject to the imminent proof that *DUsedBy* is a Well Founded relation.

In translating the recursion schema we have used $G(f, t)$ where Enderton uses $\gamma(f, t, z)$. *G* is a class function, defined for all sets; γ is a Wff, where *f, t, z* are free in γ and *z* is uniquely determined whenever all the other free variables of γ are fixed. We believe that using *G* makes the workings of the recursion theorem clearer. Typically, γ is used only to show that a recursive definition is proper.

Two translations of the recursion schema are given. The second defines *G* by cases in a way that is occasionally convenient. In particular, it is used in the definition of *DCart*. Notice that in Feature Notation we often work with functions that have a domain and codomain. To say that a function is unique when its codomain is not otherwise fixed we must declare that the codomain is as small as possible.

D : DaMod0 Properties 3

Some properties of each member of DaMod0 : Well Founded

DProp3.2 .: *DUsedBy* is a Well Founded relation

$\forall X \subseteq_d DObjs \bullet X \neq \emptyset \Rightarrow \exists t : X \bullet \forall x : X \bullet \neg\, x \; DUsedBy \; t$

DProp3.4 .: Induction principle for *DObjs* w.r.t. *DUsedBy*

$\forall X \subseteq_d DObjs \bullet [\, \forall t : DObjs \bullet DPreds(t) \subseteq X \Rightarrow t \in X \,] \Rightarrow X = DObjs$

DProp3.5 .: Well Founded induction proof schema w.r.t *DUsedBy*

Given any Wff α such that *x* is substitutable for *t* in α,

Let $\alpha' =_d \alpha^{t}{}_{x}$, then

$[\, \forall t : DObjs \bullet (\, \forall x : DPreds(t) \bullet \alpha' \,) \Rightarrow \alpha \,]$

$\Rightarrow$

$\forall t : DObjs \bullet \alpha$

Note : α must be proved unconditionally true when *t* has no immediate predecessors, i.e when $t \in DEnSets$

DProp3.6a .: Well Founded recursion theorem schema w.r.t *DUsedBy*

Given any binary class function *G* defined for all sets, then

$\exists 1\, F : \text{Function} \bullet FDef = DObjs \wedge \text{IsTotal}(F) \wedge \text{IsSurjective}(F) \wedge$

$\forall t : DObjs \bullet F(t) = G(\, \{\, \langle x, F(x) \rangle \mid x : DPreds(t) \,\}, t\,)$

DProp3.6b .: A special case

Given any unary class function G and any binary class functions G', G'' with each of G, G', G'' defined for all sets, then

$$\begin{array}{l}
\exists 1\ F : \text{Function} \bullet FDef = DObjs \wedge \text{IsTotal}(F) \wedge \text{IsSurjective}(F) \wedge \\
\quad \forall t : DObjs \bullet \\
\quad\quad \begin{array}{lll}
\text{If} & DPreds(t) = \varnothing & \\
\text{then} & F(t) = & G(t) \\
\text{else} & F(t) = & \{ G'(r, x) \mid r : t \wedge x = DConn(r) \wedge x \subseteq \text{Entities} \} \cup \\
 & & \{ G''(r, F(x)) \mid r : t \wedge x = DConn(r) \wedge x \subseteq \text{Roles} \}
\end{array}
\end{array}$$

The base members of *DObjs* are the members of *DEnSets,* the objects modelling Entity Types. Note the power of the recursion theorem. There is no requirement to define a codomain for *F* in advance. In some cases this would be difficult to do. The relation *DUses* is also Well Founded but this is a consequence of *DObjs* being finite. We do not use this property here.

To prove

Property *DProp3.2* that *DUsedBy* is a Well Founded relation. That is, to prove that

$$\begin{array}{l}
\forall D : \text{DaMod0} \bullet \\
\quad \forall X \subseteq_{\text{d}} DObjs \bullet X \neq \varnothing \Rightarrow \exists t : X \bullet \forall x : X \bullet \neg\, x\ DUsedBy\ t
\end{array}$$

We must prove for any D : DaMod0 and any non-empty subset X of *DObjs* that X contains a member none of whose immediate predecessors is a member of X. For the purposes of this proof we will say that such a member is X-minimal in D (not to be confused with Enderton's "R-minimal").

The proof is by structural induction on DaMod0. Rather than saying let α be the formula … we will say let $\alpha =_{\text{SYM}}$ ….

Let $\alpha =_{\text{SYM}} \forall X \subseteq_{\text{d}} DObjs \bullet X \neq \varnothing \Rightarrow \exists t : X \bullet \forall x : X \bullet \neg\, x\ DUsedBy\ t$

Let $\alpha' =_{\text{d}} \alpha^{D}{}_{D'}$, meaning D' is substituted for D in α.

Assume that D, D' : DaMod0.

There are three cases to consider.

C1 Case $D =_{\text{d}}$ EmpDaMod0

Then $DObjs = \varnothing$ and α is true vacuously.

C2 Case D AddedEn D'

Then by the definition of AddedEn $DObjs \subseteq D'Objs$ and $D'Objs - DObjs$ is a singleton set whose member is a set of entities.

Let $E \subseteq_{\text{d}}$ Entities be such that $\{E\} = D'Objs - DObjs$

Now consider any $X \subseteq_{\text{d}} D'Objs$. Consider two cases.

C2.C1 Case $X \neq \varnothing$ and $E \notin X$

Then $X \subseteq DObjs$. If α is true then X contains a member t that is X-minimal in D. We wish to prove that t is also X-minimal in D'. Rephrase the definition of the immediate predecessors of t to give

$$DPreds(t) =_{\text{d}} \{ x : DObjs \mid \exists r : DConnDef \bullet r \in t \wedge DConn(r) = x \}.$$

By property *DProp1.5* $DConnRan \subseteq DObjs$ and we have $DObjs \subseteq D'Objs$ so we can safely change $x : DObjs$ to $x : D'Objs$, giving

$DPreds(t) = \{ x : D'Objs \mid \exists r : DConnDef \bullet r \in t \wedge DConn(r) = x \}$.

By the definition of AddedEn, $D'Conn = DConn$ so we have

$DPreds(t) = \{ x : D'Objs \mid \exists r : D'ConnDef \bullet r \in t \wedge D'Conn(r) = x \}$.

The right-hand side is the definition of $D'Preds(t)$, therefore $DPreds(t) = D'Preds(t)$ so t is also X-minimal in D'.

C2.C2 Case $E \in X$

Then E is a member of X that is X-minimal in D' as E contains no roles and so has no immediate predecessors.

Thus if α is true then X is either empty or contains a member that is X-minimal in D'. We conclude that in case C2 if α is true then α' is true.

C3 Case D AddedRo D'

Then by the definition of AddedRo $DObjs \subseteq D'Objs$ and $D'Objs - DObjs$ is a singleton set whose member is a set of roles disjoint from any member of *DObjs*.

Let $R \subseteq_d$ Roles be such that $\{R\} = D'Objs - DObjs$

Now consider any $X \subseteq_d D'Objs$. Consider two cases.

C3.C1 Case $X \neq \emptyset$ and $X \neq \{R\}$

Let $X' =_d X \setminus \{R\}$

Then $X' \subseteq DObjs$ and $X' \neq \emptyset$. If α is true then X' contains a member t that is X'-minimal in D. We wish to prove that t is also X'-minimal in D'. As in case C2.C1 we have

$DPreds(t) = \{ x : D'Objs \mid \exists r : DConnDef \bullet r \in t \wedge DConn(r) = x \}$.

Recall that $t \langle DConn$ is the domain restriction of $DConn$ with definition domain equal to $t \cap DConnDef$. By the definition of AddedRo, $t \langle D'Conn = t \langle DConn$ (even when $t \subseteq$ Entities) provided t is disjoint from R. But $t \neq R$ and hence t and R are disjoint so we can safely change $DConn$ to $D'Conn$ giving

$DPreds(t) = \{ x : D'Objs \mid \exists r : D'ConnDef \bullet r \in t \wedge D'Conn(r) = x \}$.

The right-hand side is the definition of $D'Preds(t)$, therefore $DPreds(t) = D'Preds(t)$ so t is also X'-minimal in D'.

Finally, $DPreds(t) \subseteq DObjs$ and $R \notin DObjs$ so R is not an immediate predecessor of t. Thus whether $X = X'$ or $X = X' + \{R\}$ none of the immediate predecessors of t is a member of X so t is X-minimal in D'.

C3.C2 Case $X = \{R\}$

Then by the definition of AddedRo every immediate predecessor of R is a member of *DObjs* and so not a member of X. Thus R is a member of X that is X-minimal in D'.

Thus if α is true then X is either empty or contains a member that is X-minimal in D'. We conclude that in case C3 if α is true then α' is true.

Finally, from cases C1, C2, C3 we conclude that α is true for every member of DaMod0.

□

Notice that we can write *DConn* = *D'Conn* to say that *D* and *D'* have the same assignment of roles to objects. The use of one function to hold all role assignments, with the same domain and codomain for all members of DaMod0 (and of PreMod), has simplified the proofs.

We will now use a simple example to illustrate the use of the recursion theorem. Recall from the outline, Section 4.3.1, that we started to define the Entity Types that a Fact Type ultimately depends on. We now have the machinery to do the job properly.

Suppose we have the class function Gd, defined for all sets by

$$\begin{array}{ll} \forall a, t : \mathsf{Set} \bullet & \text{if} \quad a = \varnothing \\ & \text{then} \quad \mathsf{Gd}(a, t) =_{\mathrm{d}} \{ t \} \\ & \text{else} \quad \mathsf{Gd}(a, t) =_{\mathrm{d}} \bigcup \mathsf{Ran}(a) \quad \text{where} \;\; \mathsf{Ran}(a) =_{\mathrm{d}} \{ y \mid \exists x \bullet \langle x, y \rangle \in a \}, \end{array}$$

then we can be sure that for each *D* : DaMod0 there is a well defined function, *DDeps*, whose definition is

$$\forall t : DObjs \bullet DDeps(t) =_{\mathrm{d}} \mathsf{Gd}(\{ \langle x, DDeps(x) \rangle \mid x : DPreds(t) \}, t).$$

In practice, of course, this would be written in a way that is easier to read. For instance we could write

$$\begin{array}{ll} \forall t : DObjs \bullet & \text{if} \quad DPreds(t) = \varnothing \qquad\qquad \text{I.e } t \in DEnSets \\ & \text{then} \quad DDeps(t) =_{\mathrm{d}} \{ t \} \\ & \text{else} \quad DDeps(t) =_{\mathrm{d}} \bigcup \{ DDeps(x) \mid x : DPreds(t) \}, \end{array}$$

without defining Gd explicitly. In data modelling terms, *DDeps*(*t*) is the set of Entity Types that *t* depends on; an Entity Type depends only on itself.

We will shortly use the recursion theorem to assign cartesian products to index sets. First we define the specific representation of tuples that will be used here.

<u>*T* : Tuple</u>

The class of all fact-style tuples

∗ *TI* : Set	Index set
∗ *TDomf*	Domain function
TDomf(*i*) : (Set - {∅}) (*i* : *TI*)	Non-empty domain associated with index *i*
∗ *TVal*	Value function
TVal(*i*) : *TDomf*(*i*) (*i* : *TI*)	Value, alias element, associated with index *i*

TCond1 .: *TDomf* is a set of couples

TCond2 .: *TVal* is a set of couples

TCond3 .: *TDomfDef* = *TI* = *TValDef*

TCond1 and *TCond2* simplify several definitions, including the one immediately below.

Next we define the cartesian product operator CartProd. In Well Founded recursion we must use a function G that is defined for all sets. As CartProd may be used in the definition of G we ensure that CartProd is also defined for all sets. It returns the default value $\varnothing$ if the argument is not a set of couples forming a proper domain function. CartProd is a minor variant of the **X** operator defined in Enderton [1977], p54.

<u>CartProd</u>

Class function returning a fact-style cartesian product when given a suitable domain function.

A fact-style cartesian product is the set of all the members of Tuple that have a given index set and domain function.

Given any F then

If

(Pre 1) F is a set of couples
i.e $\exists x : \text{Set} \bullet x = F \wedge \forall y : x \bullet \exists a, b : \text{Set} \bullet y = \langle a, b\rangle$

(Pre 2) IsFunctional(F)
i.e $\forall x, y1, y2 \bullet [\langle x, y1\rangle \in F \wedge \langle x, y2\rangle \in F] \Rightarrow y1 = y2$

(Pre 3) $\neg (\varnothing \in \text{Ran}(F))$
i.e $\forall x, y \bullet \langle x, y\rangle \in F \Rightarrow y \neq \varnothing$

then

$\text{CartProd}(F) =_d \{ T : \text{Tuple} \mid TI = \text{Def}(F) \wedge TDomf = F \}$

else

$\text{CartProd}(F) =_d \varnothing$

It is convenient to give names to the nullary tuple and the nullary cartesian product.

<u>ϕ : Tuple</u>

The (unique) nullary fact-style tuple, alias 0-tuple. (ϕ is the lower case letter phi).

$\phi I =_d \varnothing$ ϕ's index set is empty

<u>$\Phi \subseteq_d$ Tuple</u>

The (unique) nullary fact-style cartesian product. (Φ is the upper case letter phi).

$\Phi =_d \text{CartProd}(\varnothing)$
$= \{ \phi \}$

Finally we define secondary features for each D : DaMod0 that assign an appropriate cartesian product to each member of *DObjs*. This is done in two stages. First, the function *DDomf* assigns a domain function to each t : *DObjs*. Second, the function *DCart* assigns the cartesian product CartProd(*DDomf*(t)) to each t : *DObjs*.

DDomf is defined by Well Founded recursion on *DObjs*. There are three cases to consider for each t : *DObjs* :

1 If $t \subseteq$ Entities then t is assigned the empty domain function $\varnothing$. Each object modelling an Entity Type is somewhat arbitrarily assigned the nullary cartesian product Φ.

2 If $t \subseteq$ Roles then t is assigned a domain function that associates each role $r : t$ with a domain determined by the object $x =_d$ *DConn*(r).

2.1 If $x \subseteq$ Entities then the domain is x itself. (x models an Entity Type).

2.2 If $x \subseteq$ Roles then the domain is the cartesian product that x identifies, namely CartProd(*DDomf*(x)).

These cases follow the recursion schema pattern given in *DProp3.6b*.

<u>*D* : DaMod0   Secondary features 2</u>

Some secondary features of each member of DaMod0 : Domain functions and cartesian products

DDomf	Function used to define *DCart*
DDomf(t) : Set (t : *DObjs*)	Domain function (as a set of couples) for the cartesian product associated with the object t

DDefDomf .: $\forall t$: *DObjs* $\bullet$
If *DPreds*(t) = $\varnothing$ (i.e $t \subseteq$ Entities)
then *DDomf*(t) $=_d \varnothing$

else *DDomf*(t) $=_d$ (i.e $t \subseteq$ Roles)
{ ⟨r, *DConn*(r)⟩ | $r : t \wedge$ *DConn*(r) $\subseteq$ Entities } $\cup$
{ ⟨r, CartProd(*DDomf*(*DConn*(r)))⟩ | $r : t \wedge$ *DConn*(r) $\subseteq$ Roles }

DCart	Cartesian product function
DCart(t) $\subseteq_d$ Tuple (t : *DObjs*)	Cartesian product associated with the object t

DDefCart .: $\forall t$: *DObjs* $\bullet$ *DCart*(t) $=_d$ CartProd(*DDomf*(t))

To conclude, we collect together all the tuples defined by each member of DaMod0 and by DaMod0 as a whole.

D : DaMod0   Secondary features 3

A secondary feature of members of DaMod0 : The fact-style tuples, alias facts, defined by *D*

$DFacts =_d \bigcup \{ DCart(R) \mid R : DRoSets \}$ The tuples defined by *D*

$= \bigcup (DCartRan \setminus \{\Phi\})$

Facts $\subseteq_d$ Tuple

The set of all fact-style tuples defined by the core model

Facts $=_d \bigcup \{ DFacts \mid D : \text{DaMod0} \}$

Note the possibility that Entities and Facts are not disjoint. This causes no problems. Informally, entities are used to construct data models but not databases and facts are used to construct databases but not data models.

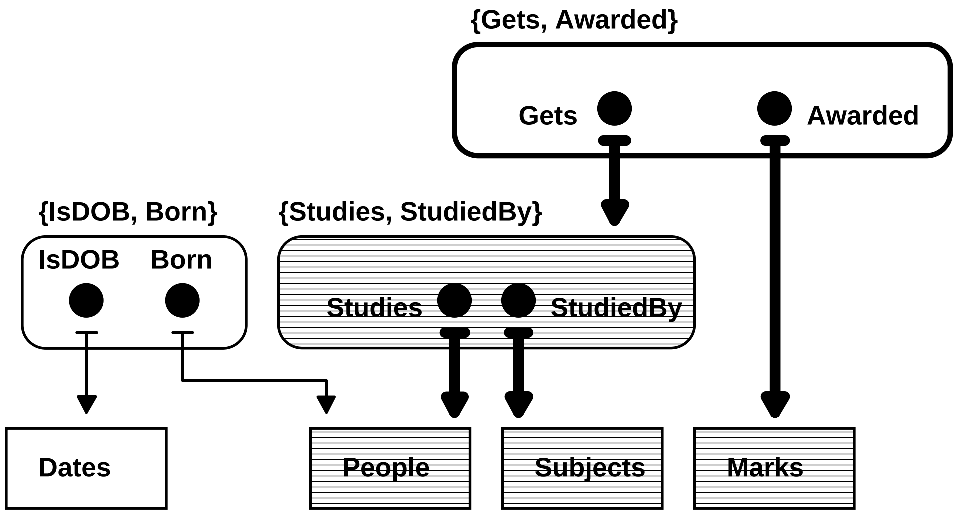
4.4 Ancestors, descendants, and rank

This is a convenient point to introduce some more secondary features of the members of DaMod0. The features will be used later on to describe pre- and post-conditions of editing operations.

4.4.1 Outline

Consider the object {Gets, Awarded} in Figure 4.4.1.1. Its immediate predecessors are {Studies, StudiedBy} and Marks. {Studies, StudiedBy} also has immediate predecessors, namely People and Subjects. We will say that an object is an **ancestor** if it is an immediate predecessor, or an immediate predecessor of an immediate predecessor, and so on. The ancestors of {Gets, Awarded} are highlighted in Figure 4.4.1.1.

Figure 4.4.1.1 Ancestors of {Gets, Awarded}



For any *D* : DaMod0 and any object *t* : *DObjs* we define the set *DAncs*(*t*) of ancestors of *t* as follows. Any object *x* : *DObjs* is an ancestor of *t*, and so a member of *DAncs*(*t*), iff

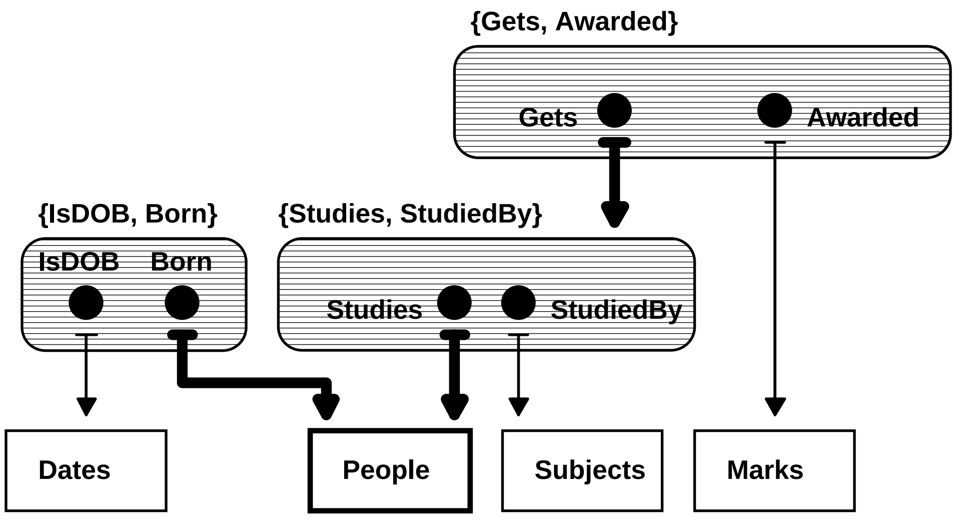
x is an immediate predecessor of *t*, or

x is an ancestor of an immediate predecessor of *t*.

This definition fits the pattern of Well Founded recursion. We can be sure that *DAncs* is well-defined for any member of DaMod0.

The opposite notion to ancestor is descendant. We will say that an object is a **descendant** if it is an immediate successor, or an immediate successor of an immediate successor, and so on. The descendants of the object People are highlighted in Figure 4.4.1.2 below.

Figure 4.4.1.2 Descendants of People



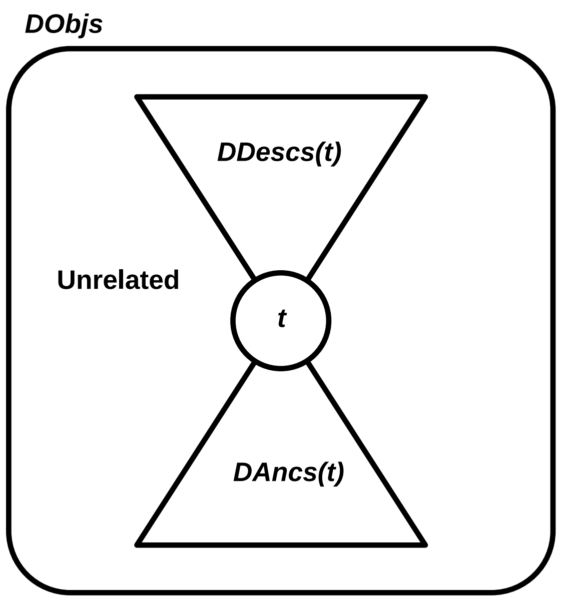
We define the set *DDescs*(*t*) of descendants of *t*, but not by recursion : *DDescs*(*t*) is defined to be the set of those members of *DObjs* that have *t* as an ancestor.

Sometimes we wish to talk of *t* and its ancestors or of *t* and its descendants. *DAncIds*(*t*) is the set of *t* and its ancestors, *DDescIds*(*t*) is the set of *t* and its descendants. To summarise, we have the four features

DAncs	Ancestors. *DAncs*(*t*) is the set of *t*'s ancestors.
DAncIds	Ancestors and self. *DAncIds*(*t*) is the set containing *t* and its ancestors.
DDescs	Descendants. *DDescs*(*t*) is the set of *t*'s descendants.
DDescIds	Descendants and self. *DDescIds*(*t*) is the set containing *t* and its descendants.

Now we will discuss some properties of ancestors and descendants. For any *t* : *DObjs* the four sets {*t*}, *DAncs*(*t*), *DDescs*(*t*), and the rest of *DObjs* are pairwise disjoint. (Some of these sets may be empty, of course). This is illustrated in Figure 4.4.1.3 below. One consequence is that no object is an ancestor of itself and conversely no object is a descendant of itself. This is to be expected. We know from the previous section that if we follow immediate predecessors from object to object we will never return to the starting point.

Figure 4.4.1.3 Quadrichotomy



Another property is also to be expected. Suppose we alter *D* to give *D'* : DaMod0 in such a way that some object *t* : *DObjs*, *t*'s ancestors, and their interconnections are unchanged. Possibly we have added some more Entity Types and Fact Types to the data model. Possibly we have removed some descendants or unrelated objects. We would expect the cartesian product, *D'Cart*(*t*), assigned to *t* to be unchanged. This is so. This is not a particularly significant property but its absence in a model of core data models would be alarming.

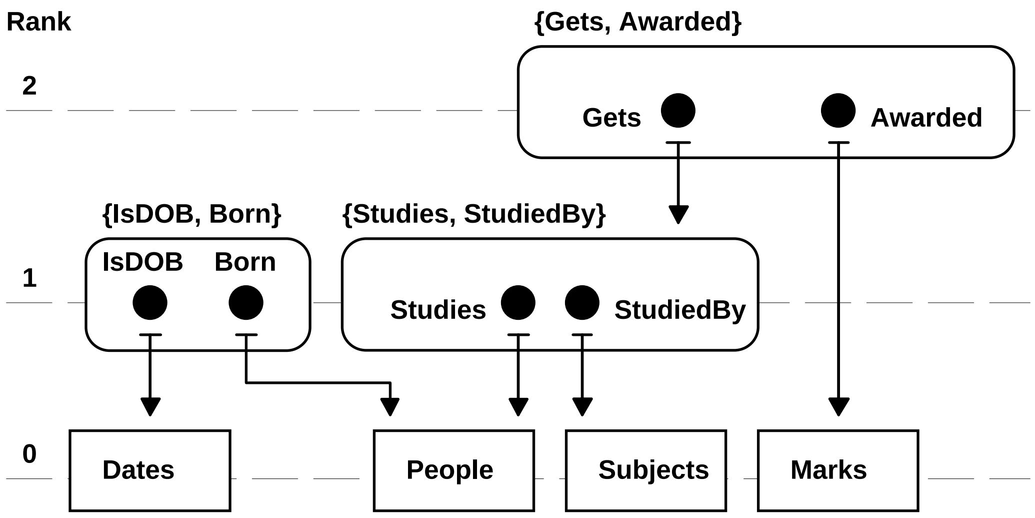
Finally we define a somewhat different feature. In the drawing style used in Figures 4.4.1.1 and 4.4.1.2 each symbol is placed as low down in the picture as possible. As a result, the object symbols form layers, with the symbols representing sets of entities belonging to the lowest layer. We can assign a number to each object that says which layer it belongs to, as in Figure 4.4.1.4 below. We will say that this number is the **rank** of the object. For each *D* : DaMod0 we define the function *DRank* that assigns a rank to each member *t* of *DObjs* as follows :

If *t* is a set of entities, and so has no immediate predecessors, then *DRank*(*t*) = 0;

Otherwise, if the highest ranking immediate predecessor of *t* has the rank *n* then *DRank*(*t*) = *n* + 1.

Once again, this definition fits the pattern of Well Founded recursion. We can be sure that *DRank* is well-defined for any member of DaMod0.

Figure 4.4.1.4 The rank of objects



Notice that any object has a higher rank than any of its ancestors and a lower rank than any of its descendants.

Notice also that objects of ranks 0 and 1 have special properties. An object of rank 0 is a set of entities. It does not identify any tuples that can be stored in the database. An object of rank 1 identifies tuples all of whose elements are primitive entities. Relational databases are designed to hold this kind of tuple. Objects of higher rank have neither of these properties.

4.4.2 Details

This section provides definitions, properties, and proofs concerning ancestors, descendants, and rank for the members of DaMod0.

We start with the definitions. Note that *DRank* has been given two definitions. The first is a general definition applicable to any Well Founded relation. With this definition the rank of an object is an ordinal. As this ordinal will be finite for any object occurring in a member of DaMod0 we give a second definition using natural numbers.

<u>*D* : DaMod0   Secondary features 4</u>

Some secondary features of each member of DaMod0 : Ancestors, descendants, and rank

$DAncs : DObjs \rightarrow \mathrm{Pow}(DObjs)$	Given $t : DObjs$ then $DAncs(t)$ is the set of ancestors of t w.r.t. *DUsedBy*

DDefAncs .:
$\forall t : DObjs \bullet DAncs(t) =_d DPreds(t) \cup \bigcup \{ DAncs(x) \mid x : DPreds(t) \}$

$DAncIds : DObjs \rightarrow \mathrm{Pow}(DObjs)$	Given $t : DObjs$ then $DAncIds(t)$ is the set consisting of t and t's ancestors

DDefAncIds .: $\forall t : DObjs \bullet DAncIds(t) =_d DAncs(t) \cup \{t\}$

$DDescs : DObjs \rightarrow \mathrm{Pow}(DObjs)$	Given $t : DObjs$ then $DDescs(t)$ is the set of descendants of t w.r.t. *DUsedBy*

DDefDescs .: $\forall t : DObjs \bullet DDescs(t) =_d \{ x : DObjs \mid t \in DAncs(x) \}$

$DDescIds : DObjs \rightarrow \mathrm{Pow}(DObjs)$	Given $t : DObjs$ then $DDescIds(t)$ is the consisting of t and t's descendants

DDefDescIds .: $\forall t : DObjs \bullet DDescIds(t) =_d DDescs(t) \cup \{t\}$

DRank	Rank function
DRank(*t*)	Rank of the object *t* w.r.t. *DUsedBy.*
(*t* : *DObjs*)	Entity Types have a rank of 0.

DDefRank1 .: $\forall t : DObjs \bullet$
$DRank(t) =_d \bigcup \{ DRank(x) \cup \{DRank(x)\} \mid x : DPreds(t) \}$

DDefRank2 .: $\forall t : DObjs \bullet$

```
If      $DPreds(t) = \varnothing$      ( i.e $t \subseteq$ Entities )
then    $DRank(t) =_d 0$
else    $DRank(t) =_d 1 + \max( \{ DRank(x) \mid x : DPreds(t) \} )$
```

We finish with a statement of some useful properties, followed by their proofs. Note that $DUsedBy^{+}$ is the transitive closure of *DUsedBy*.

<u>*D* : DaMod0 Properties 4</u>

Some properties of each member of DaMod0 : Ancestors and descendants

DProp4.4 .: $\forall t, x : DObjs \bullet x \in DAncs(t) \Leftrightarrow x\ DUsedBy^{+}\ t$
Note : $DUsedBy^{+}$ is the transitive closure of *DUsedBy*

DProp4.5 .: $\forall t, x : DObjs \bullet x \in DDescs(t) \Leftrightarrow t\ DUsedBy^{+}\ x$

Quadrichotomy

DProp4.6 .: $\forall t : DObjs \bullet t \notin DAncs(t)$

DProp4.7 .: $\forall t : DObjs \bullet t \notin DDescs(t)$

DProp4.8 .: $\forall t : DObjs \bullet DAncs(t) \cap DDescs(t) = \varnothing$

DProp4.9 .: Transitivity
$\forall x, y, z : DObjs \bullet x \in DAncs(y) \wedge y \in DAncs(z) \Rightarrow x \in DAncs(z)$

DProp4.13 .: Preservation of recursively defined values
Given any binary class functions *G*, *G'* defined for all sets, then

$$
\begin{array}{l}
\forall D' : \text{DaMod0} \bullet \forall F, F' : \text{Function} \bullet \\
\quad ([\forall t : DObjs \bullet F(t) = G(\{ \langle x, F(x)\rangle \mid x : DPreds(t) \}, t)] \wedge \\
\quad\ \ [\forall t : D'Objs \bullet F'(t) = G'(\{ \langle x, F'(x)\rangle \mid x : D'Preds(t) \}, t)] \\
\quad) \\
\quad \Rightarrow \\
\quad \forall t : DObjs \bullet \\
\qquad [\forall y : DAncIds(t) \bullet \\
\qquad\quad (y \in D'Objs) \wedge \\
\qquad\quad (y \langle DConn = y \langle D'Conn) \wedge \\
\qquad\quad (\forall a : \text{Set} \bullet G(a, y) = G'(a, y)) \\
\qquad] \\
\qquad \Rightarrow F'(t) = F(t)
\end{array}
$$

DProp4.14 .: Preservation of *DAncs* and *DCart*

$$
\begin{array}{l}
\forall D' : \text{DaMod0} \bullet \\
\quad \forall t : DObjs \bullet \\
\qquad [\forall y : DAncIds(t) \bullet y \in D'Objs \wedge y \langle DConn = y \langle D'Conn] \\
\qquad \Rightarrow \\
\qquad [D'Ancs(t) = DAncs(t) \wedge D'Cart(t) = DCart(t)]
\end{array}
$$

To prove
Property *DProp4.4* to *DProp4.9* : quadrichotomy and transitivity.

We start with *DProp4.4*.

Assume that *D* : DaMod0 and that *x*, *t* : *DObjs*. Remember that *x* is an immediate predecessor of *t* iff *x DUsedBy t*. By an obvious Well Founded induction there is a chain of immediate predecessors from t to x iff $x \in DAncs(t)$, and also iff $x\ DUsedBy^{+}\ t$. We can conclude that *DProp4.4* is true.

For *DProp4.5*, we have that

$$x \in DDescs(t) \iff t \in DAncs(x) \iff t\ DUsedBy^{+}\ x.$$

For quadrichotomy and transitivity observe that the transitive closure of any Well Founded relation is both Well Founded and a strict partial order relation (proofs in Enderton [1977], p243-245). Thus $DUsedBy^{+}$ is transitive and anti-reflexive; *DProp4.6* to *DProp4.9* are then immediate.

$\square$

To prove
Property *DProp4.13*, preservation of recursively defined values.
Assume that D, D' : DaMod0 and that F, F' : Function are defined by

$$\forall t : DObjs \bullet F(t) = G(\{ \langle x, F(x)\rangle \mid x : DPreds(t) \}, t), \text{ and}$$
$$\forall t : D'Objs \bullet F'(t) = G'(\{ \langle x, F'(x)\rangle \mid x : D'Preds(t) \}, t).$$

The proof is by Well Founded induction on *DObjs*.

Let $\beta =_{\mathrm{SYM}} \forall y : DAncIds(t) \bullet y \in D'Objs \wedge y \langle DConn = y \langle D'Conn \wedge$
$\qquad\qquad \forall a : \mathrm{Set} \bullet G(a, y) = G'(a, y)$

Let $\alpha =_{\mathrm{SYM}} [\, \beta \Rightarrow F'(t) = F(t) \,]$

Let $\alpha' =_{d} \alpha^{t}_{x}$, $\beta' =_{d} \beta^{t}_{x}$, (meaning x is substituted for t in α and β)

We wish to prove for any $t : DObjs$ that $[\, \forall x : DPreds(t) \bullet \alpha' \,] \Rightarrow \alpha$, which is to say we wish to prove that

$$[\, \forall x : DPreds(t) \bullet \beta' \Rightarrow F'(x) = F(x) \,] \Rightarrow \beta \Rightarrow F'(t) = F(t).$$

Assume that $t : DObjs$.

For any $x : DPreds(t)$ and any $y : DAncIds(x)$ we have by transitivity (*DProp4.9*) that $y \in DAncIds(t)$, so if β is true then β' is true.

Assume that $\forall x : DPreds(t) \bullet \alpha'$. Also assume that β is true, and hence that β' is true for any $x : DPreds(t)$.

Then for any $x : DPreds(t)$ we have $F'(x) = F(x)$. The value of F at t is

$$\begin{aligned} F(t) &= G(\{ \langle x, F(x)\rangle \mid x : DPreds(t) \}, t) \\ &= G(\{ \langle x, F'(x)\rangle \mid x : DPreds(t) \}, t). \end{aligned}$$

Now from β we have that $t \langle DConn = t \langle D'Conn$ so $DPreds(t) = D'Preds(t)$ and

$$F(t) = G(\{ \langle x, F'(x)\rangle \mid x : D'Preds(t) \}, t).$$

Also for any set a we have that $G(a, t) = G'(a, t)$ so

$$\begin{aligned} F(t) &= G'(\{ \langle x, F'(x)\rangle \mid x : D'Preds(t) \}, t) \\ &= F'(t). \end{aligned}$$

From this we conclude that

$$[\, \forall x : DPreds(t) \bullet \alpha' \,] \Rightarrow \beta \Rightarrow F'(t) = F(t),$$

and hence that

$$\forall t : DObjs \bullet [\, \forall x : DPreds(t) \bullet \alpha' \,] \Rightarrow \alpha,$$

and hence that $\forall t : DObjs \bullet \alpha$. From that we conclude that *DProp4.13* is true.

□

To prove
Property *DProp4.14*, preservation of *DAncs* and *DCart*.

We show that the conditions of property *DProp4.13* apply.

Assume that D, D' : DaMod0 and that $t : DObjs$.

Let $\alpha =_{\mathrm{SYM}} \forall y : DAncIds(t) \bullet y \in D'Objs \wedge y \langle DConn = y \langle D'Conn$

Let *Ga* and *Gc* be the class functions used in the recursive definitions of *DAncs* and *DCart* respectively, and *Ga'* and *Gc'* be the class functions used in the definitions of *D'Ancs* and *D'Cart*.

For *DAncs* we have $\forall a, t : \mathrm{Set} \bullet$

$$Ga(a, t) =_{\mathrm{d}} \{\, w \mid \exists \langle y, z \rangle : a \bullet w = y \,\} \cup \{\, w \mid \exists \langle y, z \rangle : a \bullet w \in z \,\}.$$

Clearly, $Ga' = Ga$, so

$$\forall y : DAncIds(t) \bullet \forall a : \mathrm{Set} \bullet Ga(a, y) = Ga'(a, y),$$

(even when α is false).

We conclude that the conditions of property *DProp4.13* apply.

For *DCart* we have $\forall a, t : \mathrm{Set} \bullet$

$$\begin{aligned} Gc(a, t) =_{\mathrm{d}} & \{\, \langle r, DConn(r) \rangle \mid r : (t \cap DRoles) \wedge DConn(r) \subseteq \mathrm{Entities} \,\} \cup \\ & \{\, \langle r, \mathrm{CartProd}(z) \rangle \mid r : (t \cap DRoles) \wedge DConn(r) \subseteq \mathrm{Roles} \wedge \\ & \qquad \langle DConn(r), z \rangle : a \,\}. \end{aligned}$$

To show that *Gc* is well defined observe that

$$\forall t : \mathrm{Set} \bullet (t \cap DRoles) \subseteq DConnDef,$$

and that CartProd is defined for all sets. To show that *Gc* has been chosen correctly observe that

$$\forall t : DObjs, r : (t \cap DRoles) \bullet DConn(r) \in DPreds(t), \text{ and}$$
$$\forall t \subseteq_{\mathrm{d}} \mathrm{Entities} \bullet Gc(a, t) = \varnothing.$$

Now if α is true then for any $y : DAncIds(t)$ we have $y \subseteq D'Roles$ and for any role $r : (y \cap DRoles)$ we have $D'Conn(r) = DConn(r)$, so

$$\forall y : DAncIds(t) \bullet \forall a : \mathrm{Set} \bullet Gc(a, y) = Gc'(a, y).$$

We conclude that the conditions of property *DProp4.13* apply.

□

Notice that the formula defining *Ga* is not one that helps us to understand the definition of *DAncs* !

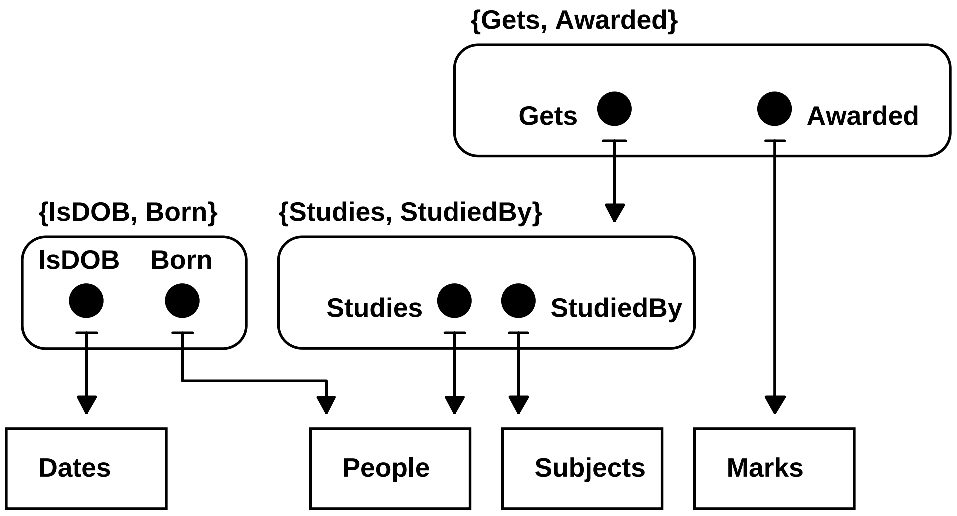
4.5 Notations to describe any *D* : DaMod0

We have shown that the members of DaMod0 have some useful mathematical properties. We should show that these properties have some practical application. Recall that one of the questions posed in Section 1.1 was "What are the essential components and structures that any computerised design tool must implement?". We will show that the definition of DaMod0 can be used to define practical ways of storing the details of a (core) conceptual data model.

4.5.1 Diagrams

We have translated NIAM diagrams into DaMod0. We should show that nothing essential has been lost. We can do this by demonstrating the opposite translation from DaMod0 back into NIAM diagrams. Suppose we are given the member of DaMod0 described in Figure 4.5.1.1.

Figure 4.5.1.1 A member of DaMod0



Let us call it Dx and describe it in a more formal manner as follows.

Dx : DaMod0 such that

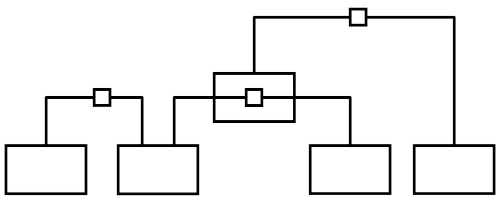
DxObjs $=_d$ { People, Subjects, Marks, Dates,
{Studies, StudiedBy}, {Gets, Awarded}, {IsDOB, Born} }

DxConn $=_d$ (Gets $\mapsto$ {Studies, StudiedBy}, Awarded $\mapsto$ Marks,
Studies $\mapsto$ People, StudiedBy $\mapsto$ Subjects,
IsDOB $\mapsto$ Dates, Born $\mapsto$ People)

where
People, Subjects, Marks, Dates $\subseteq$ Entities, and
Studies, StudiedBy, Gets, Awarded, IsDOB, Born $\in$ Roles

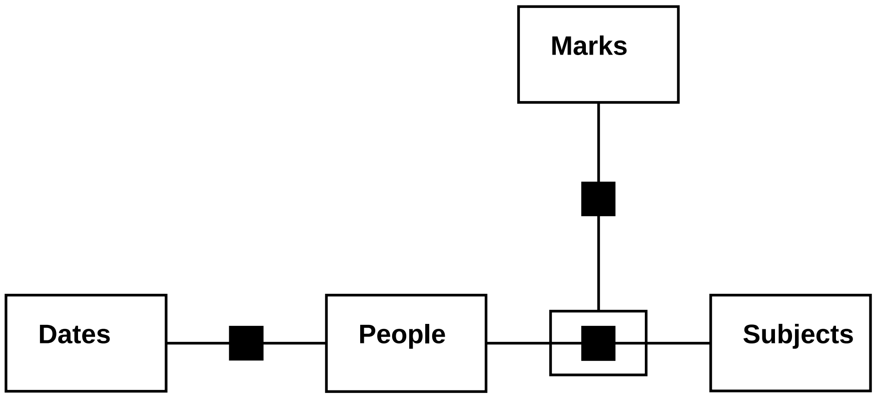
Now let us translate Dx into a picture. Translate each member *t* of DxObjs into a rectangle or square : if *t* is a set of entities then draw a distinct large rectangle; if *t* is a set of roles then draw a distinct small square. Translate each couple (*r*, *x*) of the graph of DxConn into a distinct line. Start the line at the square representing the object containing the role *r*. End the line at the square or rectangle representing the object *x*. Rather than use an arrowhead to indicate the direction of the line, ensure that it ends at a rectangle : now place rectangles around squares as necessary to do this. The result of translating Dx into a picture might look like Figure 4.5.1.2.

Figure 4.5.1.2 A picture of Dx



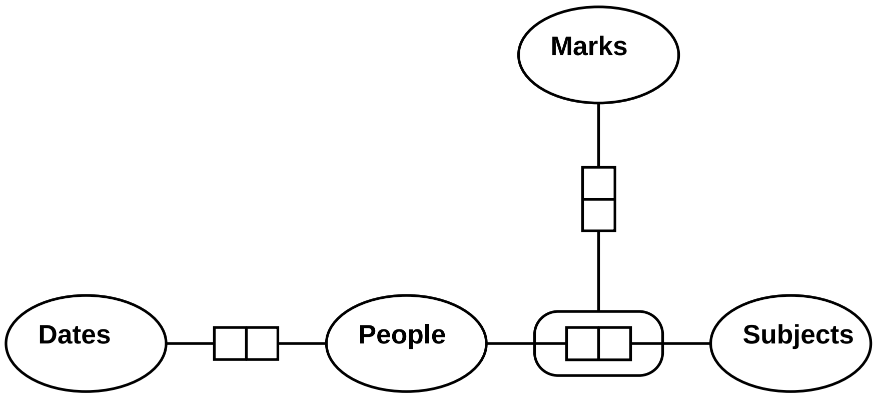
The result is not very readable. Let us rearrange the picture and add a small amount of annotation (which, remember, is not part of a core data model) to give Figure 4.5.1.3.

Figure 4.5.1.3 A more normal layout, with some annotation



We now have a conceptual data model that uses the "UMIST" dialect of the NIAM notation. If we use ellipses instead of rectangles and use sequences of boxes, one box per role, instead of squares we have a data model that uses the "original" dialect, Figure 4.5.1.4.

Figure 4.5.1.4 Using a different dialect



Clearly, any member of DaMod0 can be translated into a picture in this way. A design tool holding a representation of a member of DaMod0 has the essential information needed to draw the core part of a data model.

Notice that this kind of translation rule could be used to specify any reasonable dialect of the NIAM notation.

4.5.2 Design databases

A CASE tool used for building a conceptual data model must hold its information in some form, possibly in several forms. A likely choice for the long term storage of this data model is a relational database. Let us design such a database using DaMod0 as a guide. For simplicity we will assume that the database holds a description of one conceptual data model. We will restrict ourselves to the core part of the data model. (In practice there would be more information, and possibly several data models).

Suppose that the core data model we are interested in is modelled by some member *D* of DaMod0. We will base our database design on the constituents of *D*. Recall that *D* has the two primary features *DObjs* and *DConn*. We will require the database to hold information items that correspond to these two features. For *DObjs* the database will hold information items of the generic form

'The object *t* is a member of *DObjs*', alias '$t \in DObjs$',

where the variable *t* is restricted to the set Objects. For *DConn* the database will hold information items of the generic form

'The role *r* is associated with the object *x*', alias '$DConn(r) = x$',

where the variable *r* is restricted to the set Roles, and *x* to Objects.

We also require the database to remind us of which roles are members of which objects, so the database will hold information items of the generic form

'The role *rm* is a member of the object *xm*', alias '$rm \in xm$',

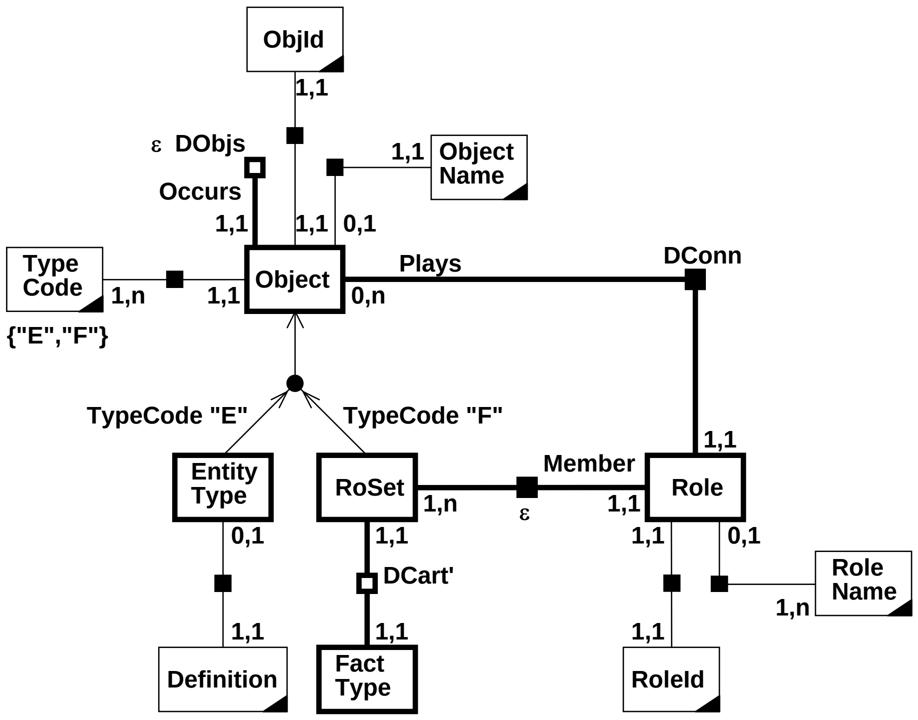
where the variable *rm* is restricted to the set Roles, and *xm* to Objects.

These generic items describe the kernel of our database. We need to add more generic items to attach concrete labels, such as text and numbers, to the abstract sets making up *D*. Recall that a data model specifies a conceptual database and that the part of the conceptual database that is to be implemented explicitly we call the actual database. The actual database will hold information items encoded as tuples, but only those tuples whose elements can be represented in the recording medium.

The resulting design is given in Figure 4.5.2.1 below. The part of the design derived directly from the definition of DaMod0 is highlighted with heavy lines. The diagram uses a Subtype symbol to say that each object belonging to *DObjs* is either an Entity Type with an optional definition, or a set of roles identifying a distinct Fact Type, but not both. This is a constraint symbol and so is not represented in the core model.

We could say that Figure 4.5.2.1 is a meta-model but this is somewhat misleading. It is an ordinary data model that specifies ordinary database instances in the usual way. One of the instances is an alternative representation of Figure 4.5.2.1. Which is the meta-model, Figure 4.5.2.1 or that database instance?

Figure 4.5.2.1 Conceptual data model specifying a database that holds one member of DaMod0



We should ask whether there is a member of DaMod0 that describes the core part of Figure 4.5.2.1. Unfortunately this is an example of the awkward cases discussed in Chapter 3 (Section 3.2.3, Example 13). The diagram includes an Entity Type named "Object" which in DaMod0 would be modelled by a set of entities. But this set is presumably modelling the set Objects that includes all subsets of Entities. It is well known that Objects is therefore strictly larger than Entities, (see, for instance, Enderton [1977], p132-133), so no set of entities can model all members of Objects.

This causes no problems in practice. We know that in a practical database design there must be a definition of each Entity Type, a definition that can be read by human beings. (For if not, the database cannot be understood by its users). We need as many distinct entities as there are possible definitions but no more. At worst there is a countably infinite set of definitions. Thus we can say that "Object" is the set of all Entity Types that will be used in practice by data modellers, knowing that this can be modelled by a subset of Entities. (Although the size of Entities has not been defined we will presume it is reasonably large). These considerations need not deter implementers.

To demonstrate that DaMod0 can indeed be used to guide the design of a practical database we have implemented Figure 4.5.2.1 as a relational database and populated it with a description of the core part of the figure. The translation of the data model into a database schema is a straightforward application of the Rmap algorithm in Halpin [1995]. The contents of the two database tables are presented in Figure 4.5.2.2 below.

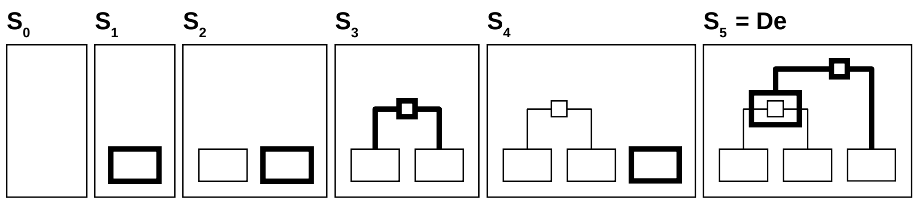
Figure 4.5.2.2 Two database reports

```
 2-Sep-1998      All objects                         Page   1

ObjId  Type    Obj Name              Entity Type definition
-----  ----  -------------------   -------------------------
E1       E   Object                All objects
E2       E   ObjId
E3       E   Type Code             {"E", "F"}
E4       E   Object Name
E5       E   Definition
E6       E   Role                  All roles
E7       E   RoleId
E8       E   Role Name
E9       E   Fact Type             All Fact Types
F1       F
F10      F   Is Member Of DObjs
F2       F
F3       F
F4       F
F5       F   DCart'
F6       F   Is Member Of
F7       F   DConn
F8       F
F9       F



31-Mar-1998       All roles                          Page   1

RoleId  RoSet  PlayedBy  Role Name
------  -----  --------  --------------------------------
R1      F1       E1
R10     F5       E9
R11     F8       E6
R12     F8       E7
R13     F6       E6      Member
R14     F6       E1
R15     F7       E6
R16     F7       E1      Plays
R17     F9       E6
R18     F9       E8
R19     F10      E1      Occurs
R2      F1       E2
R3      F2       E1
R4      F2       E3
R5      F3       E1
R6      F3       E4
R7      F4       E1
R8      F4       E5
R9      F5       E1
```

4.5.3 Construction sequences (Outline)

The definition of DaMod0 was based on the observation that a typical data model is built one step at time; each increment adds one Entity Type or Fact Type to the evolving data model and each step is a well-formed data model in its own right. Suppose we keep a record of these successive steps. If we record only the core part of each data model we would have a sequence like that shown in Figure 4.5.3.1 below. We will call such a sequence a **construction sequence**.

Figure 4.5.3.1 A construction sequence

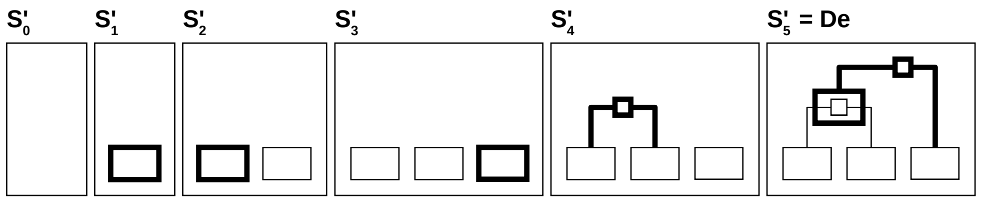


In terms of our core model a construction sequence is a sequence *S* of members of DaMod0. The first element S_0 of the sequence is the empty model EmpDaMod0. The last element is the one desired. In Figure 4.5.3.1 the last element is De. Successive elements are formed by the "proper" addition of one object. Thus successive elements S_i, S_{i+1} are related either by S_i AddedEn S_{i+1}, when a set of entities is added, or by S_i AddedRo S_{i+1}, when a set of roles with their connections is added.

Enderton defines construction sequences for any inductively generated set (Enderton [1972], p22-25), but we are more restrictive than Enderton : here each sequence must be as short as possible and successive elements of the sequence must be related.

There can be more than one construction sequence for the same data model. Figure 4.5.3.2 shows an alternative way that De could have been constructed.

Figure 4.5.3.2 An alternative construction sequence



We might wonder if construction sequences can be formed retrospectively. They can, and the procedure to do so is straightforward. Given any *D* : DaMod0 we start with the empty model EmpDaMod0 and then add the members of *DObjs* one by one until *D* is reached. Of course, we must pick the members with some care. We can pick a set of entities at any point in the sequence, but a set of roles must not be picked until all its immediate predecessors have already been picked. It is a property of every member of DaMod0 that among any set of unpicked objects there will always be at least one that can be picked next. To use different words, if the objects are displayed as a sequence in picking order then the result is a topological sort of *DObjs*, and such a sort is always possible.

We could use sequences of objects in picking order to model core data models. For instance, De in Figure 4.5.3.1 could be defined to be the sequence

People
Subjects
{Studies, StudiedBy} with (Studies $\mapsto$ People, StudiedBy $\mapsto$ Subjects)
Marks
{Gets, Awarded} with (Gets $\mapsto$ {Studies, StudiedBy}, Awarded $\mapsto$ Marks),

but the existence of alternative construction sequences could be irritating. More usefully, a CASE tool could store data models in this form. This format has the advantage that a check for the data model being well-formed can be done incrementally, object by object. It could be used as a system-independent format for the interchange of data models between CASE tools.

Retrospectively formed construction sequences have a more immediate use. They play a vital part in one of the proofs in Chapter 5 (MoveProp4, Section 5.2.2).

4.5.4 Construction sequences (Details)

Construction sequences will be used in proofs so they are defined in detail. We will be more general. A construction sequence starts at the empty model EmpDaMod0. We will allow the sequences to start at any convenient member of DaMod0 and will call them **completion sequences**. Construction sequences are then special cases.

First we need a way to model sequences in general. We will use the class ListV of all valued lists given in Scheurer [1994], p433-468. In outline, any list *L* : ListV is modelled as a value function *LV* from a set *LPts* of points to a set *LVals* of values; the points have a successor function *LS* that is constrained in the way appropriate to linear sequences. A concise version of Scheurer's definition is given here.

<u>*L* : ListV</u>

Class of all valued lists, possible empty or infinite

∗	*LPts* : Set	Index set of L; set of "points"
∗	*LVals* : Set	Value set
	LCoFam1 .: $LVals \neq \emptyset$	
∗	*LV* : $LPts \to LVals$	Value function
∗	*LS* : $LPts \mathrel{+\!\!\to} LPts$	Successor function
	LFst : *LPts*	First point of *L* (defined iff $LPts \neq \emptyset$)

LCo1 .: $LPts \neq \emptyset \;\Rightarrow\; LFst \in LPts - LSRan$

LCo2 .: IsInjective(*LS*)

LCo3 .: Induction principle
$\forall A \subseteq_d LPts \;\bullet$
$[\, LFst \in A \;\wedge\; \forall i : (A \cap LSDef) \;\bullet\; LS(i) \in A \,] \;\Rightarrow\; A = LPts$

LLst : *LPts* - *LSDef*	Last point of *L* (defined iff *LPts* is finite)

Notice for any *L* : ListV that *LS* is a Well Founded relation. If *L* is non-empty then *LFst* is the only base point and any other point has exactly one immediate predecessor.

We wish to define the function CompSeq which returns a list of members of DaMod0 when given the member(s) of DaMod0 that are to be the first and last elements of the list. The function is required to ensure that successive elements of the list are related by AddedEn or AddedRo. Thus successive elements differ by one object. We will use a choice function to determine this object. (More than one object can be eligible). The choice function will be given to CompSeq as an argument.

To simplify the pre-conditions for CompSeq we will require the choice function to be defined for any set of objects of any member of DaMod0. That is, it must be a choice function on the set Objects. We define the set ObChos of all such choice functions. We will assume that ObChos is not empty. Although this relies on the Axiom of Choice the functions will always be applied to finite sets of objects so this should not be contentious.

ObChos : Set

All possible choice functions for the set Objects

ObChos $=_d$
$\{ C : \text{Pow(Objects)} \rightarrow \text{Objects} \mid \forall X \subseteq_d \text{Objects} \bullet X \neq \emptyset \Rightarrow C(X) \in X \}$

Choice functions have been defined as total functions which is somewhat unconventional but convenient. For any C : ObChos we can regard $C(\emptyset)$ as an arbitrary member of Objects.

Now we can define the function CompSeq that returns a completion sequence when given a choice function, a start model, and an end model. The start model must be a "sub-model" of the end model in the sense defined by the function's pre-conditions. CompSeq returns a member of ListV. Note that the list is not fully determined : there is a free choice of the points used in the list. The definition is followed by a proof that the function is well defined; a list meeting the requirements always exists and is unique up to isomorphism.

CompSeq : Function

Function which when given any C : ObChos, De : DaMod0, and a suitable Ds : DaMod0 returns a completion sequence that starts at Ds and ends at De

Given any C : ObChos, Ds, De : DaMod0 with

(Pre 1) $DsObjs \subseteq DeObjs$

(Pre 2) $DsConnGr \subseteq DeConnGr$

then

$\text{CompSeq}_C(Ds, De) =_d L$ where L : ListV and

$LVals =_d \text{DaMod0} \wedge$ (a)

$LV(LFst) =_d Ds \wedge$ (b)

$LV(LLst) =_d De \wedge$ (c)

$[\ \forall D : LVRan \bullet$
$DObjs \subseteq DeObjs \wedge$ (d)
$DConnGr \subseteq DeConnGr$ (e)
$] \wedge$

```
[ $\forall i$ : $LSDef$ $\bullet$
    Let $D =_d LV(i)$, $D' =_d LV(LS(i))$
    Let $t =_d C(\{ x : (DeObjs - DObjs) \mid DePreds(x) \subseteq DObjs \})$

    $D'Objs =_d DObjs + \{t\}$ $\wedge$              (f)
    $D'ConnDef =_d D'Roles$                (g)
]
```

ConsSeq : Function

Function which when given any *C* : ObChos and a *D* : DaMod0 returns a construction sequence for *D*

Given any *C* : ObChos, *D* : DaMod0 then

$\text{ConsSeq}_C(D) =_d \text{CompSeq}_C(\text{EmpDaMod0}, D)$

To Prove

That CompSeq is well-defined. That is, for any *C*, *Ds*, *De* satisfying the preconditions there exists a list *L* : ListV obeying the definition of $\text{CompSeq}_C(Ds, De)$, and *L* is unique up to isomorphism.

Assume that *C* : ObChos, *Ds*, *De* : DaMod0, and that Pre 1 and Pre 2 are true.

First, isomorphism.

Assume that a list *L* : ListV obeying the definition of $\text{CompSeq}_C(Ds, De)$ exists. The proof is by induction on *LPts*. We wish to prove that the first value of *L* and all successive values are uniquely determined independently of *LPts*.

The first value is *Ds*, by definition, and so is uniquely determined.

Let *D*, *D'* be successive values as in the definition of CompSeq. Assume that *D* is uniquely determined. Then by (f) *D'Objs* is uniquely determined and hence so is *D'Roles*, the roles occurring in *D'*. Therefore by (g) and (e) *D'Conn* is also uniquely determined and so *D'* is uniquely determined. This value *D'* is independent of the choice of points.

By (c) we have that the list ends at an occurrence of the value *De*. We wish to prove that it must end at the first occurrence. Suppose it does not so there is a point *i* : *LSDef* for which $LV(i) = De$. Let *D'* be the next value, $LV(LS(i))$. By (f) we have $D'Objs = DeObjs + \{t\}$ where $t = C(\emptyset)$. By (d) we have $t \in DeObjs$. But the use of the "+" symbol in (f) signals the union of disjoint sets, so we have $t \notin DeObjs$. We conclude that $i \notin LSDef$, so the list must end here. This conclusion is independent of the choice of points.

We can conclude that any lists obeying the definition of $\text{CompSeq}_C(Ds, De)$ are isomorphic.

Second, existence.

Clearly, there are lists of values belonging to PreMod that obey items (b), (d), (e), (f), and (g) of the definition. It suffices to prove for any such list L : ListV that is as long as possible that the following are also true :

E1) The value *De* occurs at least once;

E2) Each value is a member of DaMod0.

E1

Observe that in item (f) of the definition of CompSeq we have a choice function C acting on a subset of *DeObjs*. As *DeUsedBy* is a Well Founded relation we can be sure that there is an x : (*DeObjs* - *DObjs*) such that *DePreds*(x) $\subseteq$ *DObjs* whenever *DObjs* $\subset$ *DeObjs*. Thus C chooses each member of *DeObjs* - *DsObjs* in turn exactly once and there is no reason to end the list until *De* is reached. As *DeObjs* is finite *De* will be reached in a finite number of steps. We can conclude that the value *De* occurs at least once.

E2

The proof is by induction on *LPts*. Recall that two of the generators of DaMod0 are the relations AddedEn and AddedRo.

LV(*LFst*) = *Ds* and *Ds* is a member of DaMod0 by definition.

Now suppose that i : *LSDef* and that $D =_d LV(i) \in$ DaMod0, $D' =_d LV(LS(i)) \in$ PreMod. By item (f) of the definition of CompSeq we have *D'Objs* = *DObjs* $\cup$ {t} where t : *DeObjs* and $t \notin$ *DObjs*. By the properties of *De* we have that t is a non-empty member of Objects disjoint from each member of *DObjs*. We have two cases to consider.

E2.C1 Case $t \subseteq$ Entities

Then *D'Roles* = *DRoles* so by (g) and (e) of the definition we have *D'Conn* = *DConn*. Thus D' meets the conditions for D AddedEn D' and so $D' \in$ DaMod0.

E2.C2 Case $t \subseteq$ Roles

Then by the properties of *De* we have that t is a finite set of roles. We also have *D'Roles* = *DRoles* + t. By (g) of the definition of CompSeq we have *D'ConnDef* = *DConnDef* + t. By (e) we have that for any r : *D'Roles* either $r \in t$ so by the choice of t we have *D'Conn*(r) $\in$ *DObjs*, or $r \notin t$ and *D'Conn*(r) = *DConn*(r). Thus D' meets the conditions for D AddedRo D' and so $D' \in$ DaMod0.

We can conclude that for any i : *LPts*, *LV*(i) $\in$ DaMod0.

Finally, from E1 and E2 we can conclude that there is a list L : *ListV* that obeys the definition of CompSeq. Altogether, we can conclude that CompSeq is well defined (up to isomorphism).

Observe that completion sequences enable DaMod0 to be treated as a category; that is, as a model of the axioms given in Mac Lane[1971], p7-8. This category, which we will call DMComp, is defined by :

- Objects : The members of DaMod0.
- Arrows : The completion sequences from member to member.
- Identity arrows : The one-element completion sequences.
- Composition : Concatenation of completion sequences, defined in the obvious way.

The arrows in the opposite category DMComp^{op} are, of course, the (possibly partial) destruction sequences. However, we have not found a use for these categories so far.

5 Core model : Operations

This chapter continues the job of building a set-theoretical model of "all" well-formed NIAM conceptual data models. The previous chapter concentrated on definitions and on the properties of each model of a data model. This chapter concentrates on operations that model the conversion of one data model into another, and on equivalence relations between data models. We are still restricting ourselves to the core part of data models here.

We will define two kinds of operation that transform one data model into another. Well-formed data models have been defined to be ones that can be built incrementally, starting with the empty data model and adding one Entity Type or Fact Type at a time in a proper manner. The first kind of operation we define are operations that perform these primitive actions (Section 5.1). The second kind do changes of a more global nature : copy or erase part of a data model, merge two data models together, change a role connection (Section 5.2). Together these operations form a small convenient basis from which all desired operations can be composed. Of practical importance is to know when applying one of these operations to a well-formed data model will result in another well-formed data model. For each operation we will describe and prove the necessary preconditions.

One of the global operations enables us to extract a description of any Fact Type and display it in isolation (Section 5.3). We could then describe any data model as the merging of many such Fact Type descriptions if we wished.

The model uses the members of the sets Entities and Roles as arbitrary placeholders. Obviously, the precise choice of roles or entities when modelling a given data model should not make any significant difference. We will define an equivalence relation that expresses this (Section 5.4).

A common occurrence in data modelling is to find that a business requirement can be described in several different ways in a data model. Possibly one way is clearer to readers, another way leads to a more efficient database implementation. It is important to know when two constructions do essentially the same job. Many cases of equivalent constructions have been published; we will make only a passing reference to these. However, one case that appears to have had no general treatment is constructions using objectified Fact Types. That is, when Fact Types have domains that are also Fact Types. We will define an equivalence relation that shows when two Fact Types are able to describe the same business requirement (Section 5.5).

There is a special case of this equivalence. A typical relational database can store tuples whose elements are entities, but not tuples whose elements are tuples defined in a data model. We prove that any Fact Type can be "flattened" to give an equivalent Fact Type whose domains are all Entity Types. Thus objectified Fact Types can be used in a data model without prejudicing implementation.

As in the previous chapter the text has been split into outline parts that give a less formal description and detail parts that give the full mathematics.

5.1 Incremental editing operations

Recall that core data models are modelled by the members of the set PreMod and that DaMod0 is the subset of PreMod whose members are deemed to be well-formed. DaMod0 is defined by means of the generators EmpDaMod0, AddedEn, and AddedRo. These three define the pre- and post-conditions for creating the empty model, the "proper" addition of one set of entities, and the "proper" addition of one set of roles, respectively. We now wish to define three editing functions that perform these actions.

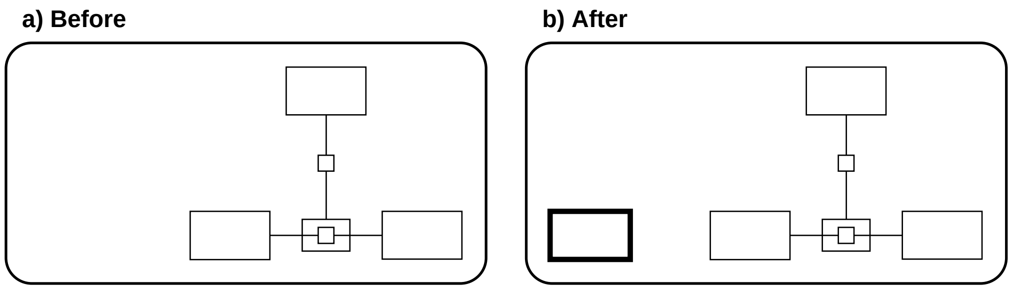
5.1.1 Outline

We wish to model the operation that gives us an initial empty data model, as in Figure 5.1.1.1;

Figure 5.1.1.1 Start a new conceptual data model

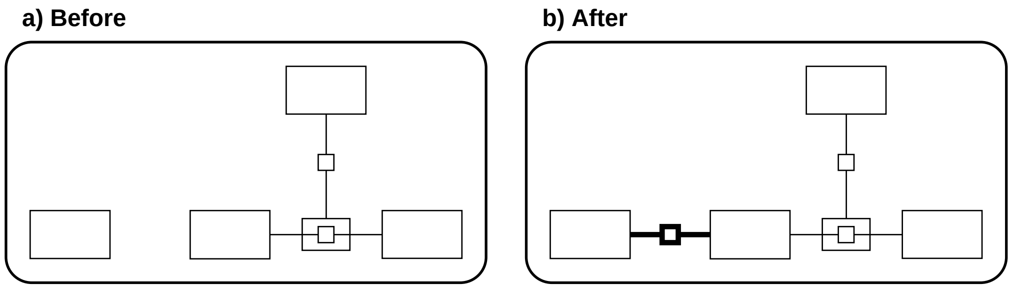
and the operation that adds one Entity Type to a data model, as in Figure 5.1.1.2;

Figure 5.1.1.2 Add one Entity Type



and the operation that adds one Fact Type to a data model, as in Figure 5.1.1.3.

Figure 5.1.1.3 Add one Fact Type



Although we are mostly interested in the members of DaMod0 we will be more general and will define editing functions that can be applied to any member of PreMod. This allows us to use the functions inside composite functions where intermediate states may be ill-formed. It also allows us to use the functions in proofs at points where the operand is yet to be proven a member of DaMod0. The preconditions that ensure a member of DaMod0 is transformed into a member of DaMod0 are thus a property to be stated separately.

Defining a function to create the empty model is trivial. We already have the constant EmpDaMod0, the member of PreMod with no objects and no role associations. This constant can be described as a function taking no arguments that returns a member of PreMod. By definition, EmpDaMod0 is a member of DaMod0.

Next we define the function AddEnTy that adds any set *E* of entities to any member *P* of PreMod. There are no preconditions. Recall that the two primary features of any member of PreMod are a set of objects and a function associating roles with objects. The objects of AddEnTy(*P*, *E*) are *E* and the objects of *P* (which might already include *E*). The role associations are unchanged.

If *P* is a member of DaMod0 and *E* obeys the conditions given in the definition of AddedEn (Section 4.2.3) then AddEnTy(*P*, *E*) is sure to be a member of DaMod0.

Finally we define the function AddFaTy that adds a set of roles and some role associations to any member *P* of PreMod. The role associations are given as any partial function *C* from roles to objects. As the usual target of AddFaTy is the members of DaMod0 where all roles are associated with objects we do not use a separate argument for the set of roles to be added; we use the definition domain *CDef* of *C*. There are no preconditions. The objects of AddFaTy(*P*, *C*) are *CDef* and the objects of *P*. The role associations are those of *C* and those of *P* where not overridden by *C*.

If *P* is a member of DaMod0 and *CDef* and *C* obey the conditions given in the definition of AddedRo (Section 4.2.3) then AddFaTy(*P*, *C*) is sure to be a member of DaMod0.

5.1.2 Details

This section defines the operations on PreMod that describe incremental changes to data models.

First we repeat the definition of EmpDaMod0 and state a relevant property.

EmpDaMod0 (Repeated definition)

Nullary function, alias constant, that returns the (unique) empty core model

EmpDaMod0 $=_d$ *P′* where *P′* : PreMod and

P′Objs $=_d \varnothing$	The model contains no objects
P′ConnDef $=_d \varnothing$	No roles are associated with objects

---

EmpDaMod0 Properties 1

A property of the nullary function EmpDaMod0

EmpDaMod0Prop6 .: EmpDaMod0 $\in$ DaMod0

The property is immediate as EmpDaMod0 is defined to be a member of DaMod0.

Next we define the function AddEnTy and then state the preconditions for remaining within DaMod0.

AddEnTy

Function on members of PreMod : add one set of entities.
Model the addition of one Entity Type to a data model.

Given any P : PreMod, $E \subseteq_d$ Entities then

AddEnTy(P, E) $=_d$ P' where P' : PreMod and

$P'Objs =_d PObjs \cup \{E\}$	Add the set of entities
$P'Conn =_d PConn$	Role associations are unchanged

AddEnTy Properties 1

A property of the function AddEnTy

AddEnTyProp1 .:
$\forall D$: DaMod0 $\bullet$ $\forall E \subseteq_d$ Entities $\bullet$
[$E \neq \varnothing \wedge (\forall x : DObjs \bullet x \cap E = \varnothing)$] $\Rightarrow$ AddEnTy(D, E) $\in$ DaMod0

The property is immediate : the conditions ensure that D AddedEn AddEnTy(D, E).

Finally we define the function AddFaTy and the additional preconditions for remaining within DaMod0.

<u>AddFaTy</u>

Function on members of PreMod : add one set of roles and their role associations. Model the addition of one Fact Type to a data model (when well formed).

Given any P : PreMod, C : Roles $+\!\!\rightarrow$ Objects then

AddFaTy(P, C) $=_d$ P' where P' : PreMod and

$P'Objs =_d PObjs \cup \{CDef\}$		Add the set of roles
		Override role associations
$CDef \langle P'Conn$	$=_d CDef \langle C$	
$(\text{Roles} - CDef) \langle P'Conn$	$=_d (\text{Roles} - CDef) \langle PConn$	

<u>AddFaTy Properties 1</u>

A property of the function AddFaTy

AddFaTyProp1 .:

$\forall D$: DaMod0 $\bullet$ $\forall C$: Roles $+\!\!\rightarrow$ Objects $\bullet$
[$CDef \neq \varnothing$ $\wedge$ IsFinite($CDef$) $\wedge$
($\forall x : DObjs \bullet x \cap CDef = \varnothing$) $\wedge$
($\forall r : CDef \bullet C(r) \in DObjs$)]
$\Rightarrow$
AddFaTy(D, C) $\in$ DaMod0

The property is immediate : the conditions ensure that D AddedRo AddFaTy(D, C).

5.2 Global editing operations

Recall from Section 4.5.3 that for every member *D* of DaMod0 there is a construction sequence. That is, a sequence of members of DaMod0 starting with EmpDaMod0, ending with *D*, and such that successive elements of the sequence are related either by AddedEn or by AddedRo. We can conclude that any member of DaMod0 can be constructed by repeated use of the three functions EmpDaMod0, AddEnTy, and AddFaTy. These three functions form a minimal set of operators from which any member of DaMod0 can be built. However, they are not a very convenient set of operators. If we needed to delete an Entity Type or a Fact Type we would have to erase the whole model and start again. In practice, some "cut and paste" operators will be needed. We will define four more functions that modify data models in a more general way.

5.2.1 Outline

We wish to model operations that enable us to copy a selected part of a data model, erase a designated part of a data model, merge data models together, and change role associations. As with the incremental operations we will define the operations on PreMod and will avoid preconditions wherever possible.

We wish to select a coherent set of editing functions. Is there an obvious way to choose them? It is obvious how merging should be defined : we merge objects and role associations. For copying we must select both the objects and the role associations to be copied. However, we are biased towards operations on the members of DaMod0. The role associations to be copied are determined by the selected objects : we copy a role association iff the role belongs to one of the selected objects. For erasing we must also select objects and role associations. We will ensure that we can erase that which has been copied, so that copying and erasure are complementary. We will also ensure that we can highlight differences between models. For changing role associations we will choose the simplest case : reassign one role.

We will study four functions meeting these requirements; many other choices were possible, of course. The four functions, somewhat arbitrarily named Tear, Diff, Merge, and Move, are based on the set operations intersection ($\cap$), relative complement (\ ), union ($\cup$), and function override. For each function there are preconditions, given here as properties, that ensure that members of DaMod0 are transformed into members of DaMod0. The preconditions make use of ancestors and descendants (defined in Section 4.4).

The first operator, Tear, models the extraction, copying, tearing out of a submodel from a data model. Tear could be used

To extract part of a data model for re-use in another data model;

To split a large data model into smaller models that are easier to consult;

To indicate the part of a data model that is to be highlighted or dimmed;

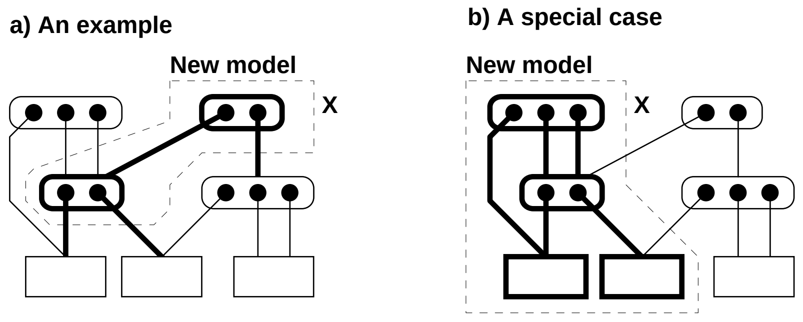
To define an application view, restricting an application or user to certain parts of the database contents;

To define the part of a database that is to be archived;

To produce the empty data model by extracting nothing (as an alternative to the incremental operator).

The action of Tear is illustrated in Figure 5.2.1.1(a) below. We have some member *P* of PreMod and a set *X* of objects. The result, Tear(*P*, *X*), is the member of PreMod highlighted in the figure. Its objects are those objects of *P* that are also members of *X*. Its role associations are those of *P* but only for roles belonging to the objects selected by *X*.

Figure 5.2.1.1 Tear



The highlighted part of Figure 5.2.1.1(a) is obviously not a member of DaMod0. Tear(*P*, *X*) has roles associated with objects that do not occur in Tear(*P*, *X*), and no member of DaMod0 is like that. However in Figure 5.2.1.1(b) the result is a member of DaMod0. There, whenever an object is selected by *X* then so are all its ancestors. (Equivalently, so are all its immediate predecessors). This is the precondition needed to ensure that a member of DaMod0 is transformed into a member of DaMod0.

Notice that *X* is allowed to be any subset of Objects; it is not restricted to the objects of *P*. This does no harm and it makes some proofs simpler.

The second operator, Diff, models the erasing or deletion of part of a data model. Diff could be used

To delete unwanted Entity Types and Fact Types;

To prepare a data model for re-use in another data model;

To hide temporarily uninteresting parts of a data model;

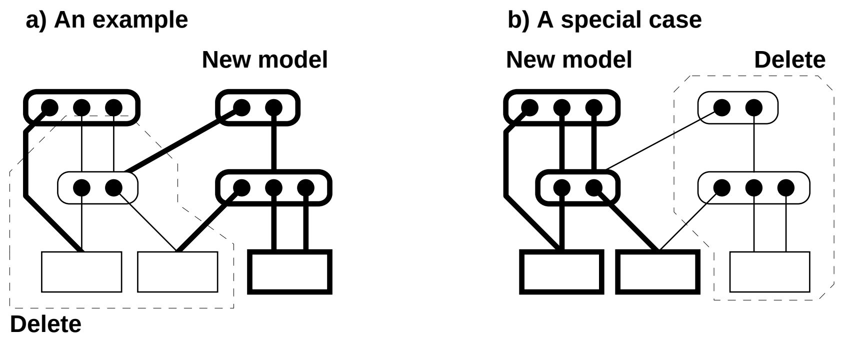
To show where one data model differs from another;

To show what has not been extracted by a Tear operation;

To define the permitted scope of variables in the formulas that define derived Fact Types and Subtypes (by removing forbidden areas).

The action of Diff is illustrated in Figure 5.2.1.2(a) below. We have some member *P* of PreMod and a means of saying which objects and role associations are to be removed. The result is the member of PreMod highlighted in the figure. A convenient structure for indicating the objects and role associations to be removed is another member of PreMod, say *P1*. Thus the highlighted part of Figure 5.2.1.2(a) is Diff(*P*, *P1*). Its objects are those objects of *P* that are not objects of *P1*. Its role associations are those of *P* that are not role associations of *P1*.

Figure 5.2.1.2 Diff



The highlighted part of Figure 5.2.1.2(a) is obviously not a member of DaMod0 but in Figure 5.2.1.2(b) the highlighted part is a member. There, the objects and role associations to be erased can be described by the result of a Tear operation. The final result is Diff(*P*, Tear(*P*, *X*)) for some set *X* of objects. In addition, whenever an object is selected for removal by *X* then so are all its descendants. (Equivalently, so are all its immediate successors). This is the precondition needed to ensure that a member of DaMod0 is transformed into a member of DaMod0.

Notice that *P1* is allowed to be any member of PreMod; it is not restricted to sub-models of *P*. This does no harm and it allows us to display differences between data models. Notice also that we have made use of a Tear operation where the result of doing Tear is unlikely to be a member of DaMod0.

The third operator, Merge, models the merging or joining together of two data models. Merge could be used

To combine together parts of other data models that are being re-used;

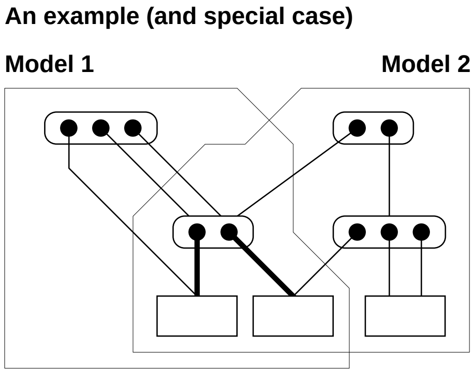
To reconstruct a data model that was split into smaller models;

To describe the mechanics of joining partial models of an organisation to form an overall model ("view integration");

To describe the addition of an Entity Type or Fact Type to a data model (an alternative to the incremental operators).

The action of Merge is illustrated in Figure 5.2.1.3 below. We have the two members *P1* and *P2* of PreMod outlined by dotted lines. The result, Merge(*P1*, *P2*), is the member of PreMod shown in the figure. Its objects are the objects of *P1* and *P2*. Its role associations are those of *P1* and *P2*. Unlike the other operators we need a precondition for Merge. We cannot allow the result to show one role associated with two different objects. Thus we require as a precondition that any role occurring in both *P1* and *P2* is associated with the same object in each. The common role associations are highlighted in the figure.

Figure 5.2.1.3 Merge



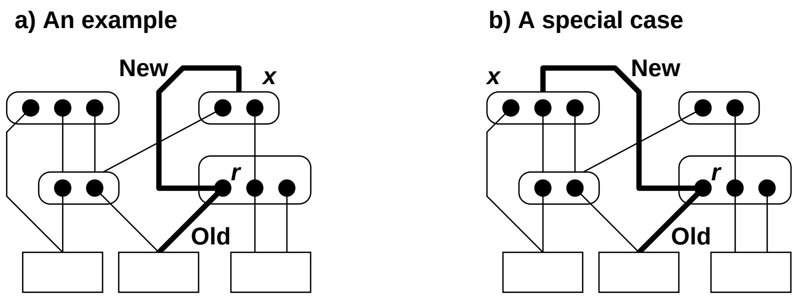
The result in Figure 5.2.1.3 might not be a member of DaMod0. There might be two objects occurring in the result that are different but not disjoint, and no member of DaMod0 is like that. However, if all the objects occurring in the result are pairwise disjoint then the result is a member of DaMod0. This is the additional precondition needed to ensure that two members of DaMod0 are transformed into a member of DaMod0.

Notice that we always have Merge(Tear(*P*, *X*), Diff(*P*, Tear(*P*, *X*))) = *P*. The operators Tear and Diff are complementary and Merge can be used to undo their actions.

The fourth and last operator, Move, models a change of role association. It could be used to change the domain of a Fact Type; more accurately, it could be used to change the Fact Type, alias cartesian product, identified by a set of roles.

The action of Move is illustrated in Figure 5.2.1.4(a) below. We have some member *P* of PreMod, a role *r*, and an object *x*. The result, Move(*P*, *r*, *x*), is the member of PreMod that is the same as *P* except that the role *r* is now associated with the object *x*. In the figure the old and new role associations are highlighted.

Figure 5.2.1.4 Move



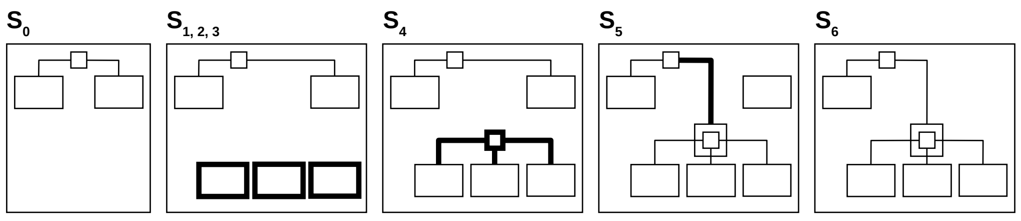
The result in Figure 5.2.1.4(a) is obviously not a member of DaMod0. In Move(*P*, *r*, *x*) the object containing *r* is now an ancestor of itself (and also a descendant of itself), and no member of DaMod0 is like that. However in Figure 5.2.1.4(b) the result is a member of DaMod0. There, *r* is a role of *P*, *x* is an object of *P*, and *x* is neither the object containing *r* nor any of its descendants. This is the precondition needed to ensure that a member of DaMod0 is transformed into a member of DaMod0.

Notice that *r* is allowed to be any member of Roles and *x* is allowed to be any member of Objects; they are not restricted to the roles and objects occurring in *P*. Notice also that unlike the other operators Move can change the rank of an object and that changes affecting different roles can be done in any order.

Many more functions could be defined but it is likely that any additional functions can be defined as the composition of those already given. We will give two examples of this. In the first example we show how an Entity Type can be transformed into a Fact Type, and vice versa.

Suppose we have a data model where colours are identified by printers' Pantone numbers, so we have an Entity Type of all Pantone numbers. Now we hear that the requirements have changed and that each colour is to be identified by a computer screen's RGB triple. We need to convert the Entity Type into a Fact Type of all RGB triples. This can be done by the sequence of changes shown in Figure 5.2.1.5 below.

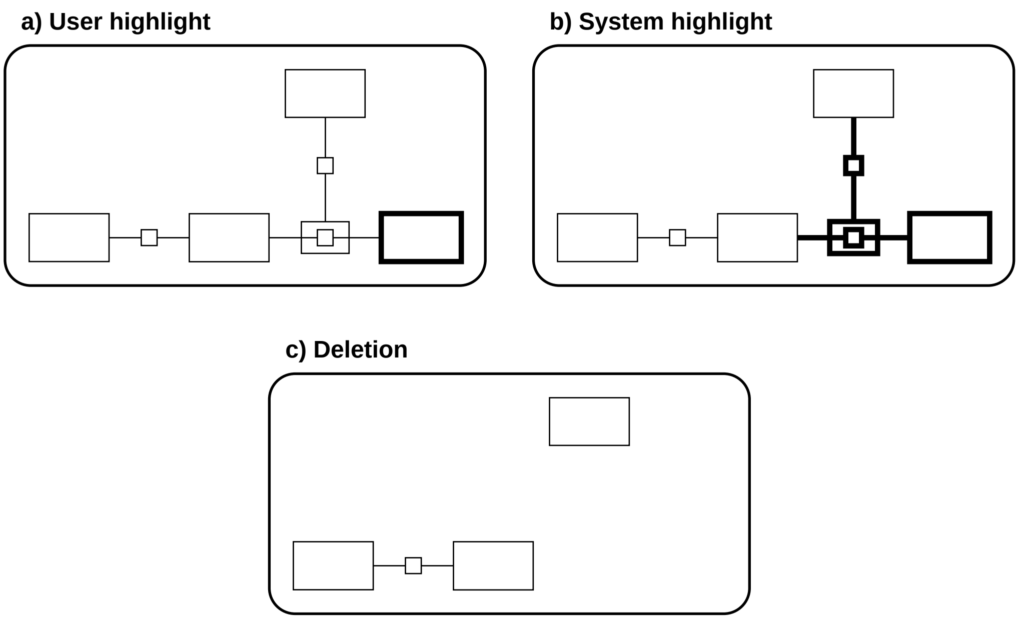
Figure 5.2.1.5 Converting an Entity Type into a Fact Type, and vice versa



Each step in the sequence can be done by one of the operators already described : add an Entity Type (three times), add a Fact Type, change a role association, delete an object. The sequence can also be reversed to convert a Fact Type into an Entity Type.

In the second example we show how an editor can help a user to delete an object in a way that leads to a well-formed data model. Suppose we have the data model shown in Figure 5.2.1.6(a) below. The user has highlighted an Entity Type and expressed a desire to delete it. The system highlights other parts of the data model that must also be deleted, Figure 5.2.1.6(b). The result of the deletion, a well-formed data model, is given in Figure 5.2.1.6(c).

Figure 5.2.1.6 A system-assisted deletion



These actions can be described as follows. Suppose the data model in Figure 5.2.1.6(a) is described by *D* : DaMod0 and the object highlighted by the user is *t*. Then the editor's highlighting in Figure 5.2.1.6(b) is described by Tear(*D*, *DDescIds*(*t*)), and the altered data model by Diff(*D*, Tear(*D*, *DDescIds*(*t*))). *DDescIds*(*t*) is the set of objects comprising *t* and all its descendants. It is closed under descendants so the result of the Diff operation is a member of DaMod0 and Figure 5.2.1.6(c) is sure to be well-formed. The minimum possible deletion has been done.

5.2.2 Details

This section provides definitions, properties, and proofs for the four global operators Tear, Diff, Merge, and Move. They are dealt with in order without any commentary.

Tear

Function on members of PreMod : Tear out a submodel

Given any *P* : PreMod, $X \subseteq_d$ Objects then

Tear(*P*, *X*) $=_d$ *P'* where *P'* : PreMod and

$P'Objs =_d PObjs \cap X$ Restrict objects

$P'ConnGr =_d PConnGr \cap (P'Roles \times Objects)$ Restrict definition domain

---

Tear Properties 1

Some properties of the function Tear

TearProp3 .:
$\forall D$: DaMod0 $\bullet$ $\forall X \subseteq_d$ Objects $\bullet$
$[\ \forall t : (DObjs \cap X)\ \bullet\ DAncs(t) \subseteq X\ ] \Rightarrow$ Tear(D, X) $\in$ DaMod0

TearProp4 .: $\forall D$: DaMod0 $\bullet$ $\forall t : DObjs$ $\bullet$ Tear(D, $DAncIds(t)$) $\in$ DaMod0

TearProp5 .: $\forall P$: PreMod $\bullet$ Tear(P, $\varnothing$) = EmpDaMod0

TearProp4 is an application of TearProp3; TearProp5 is immediate from the definition of Tear. We need prove only TearProp3.

To prove
Property TearProp3 that when a certain precondition is true then Tear transforms a member of DaMod0 into a member of DaMod0. That is, to prove that
$\forall D$: DaMod0 $\bullet$ $\forall X \subseteq_d$ Objects $\bullet$
$[\ \forall t : (DObjs \cap X)\ \bullet\ DAncs(t) \subseteq X\ ] \Rightarrow$ Tear(D, X) $\in$ DaMod0

The proof is by structural induction on DaMod0.

Let $\beta =_{\text{SYM}} [\ \forall t : (DObjs \cap X)\ \bullet\ DAncs(t) \subseteq X\ ]$

Let $\alpha =_{\text{SYM}} \beta \Rightarrow$ Tear(D, X) $\in$ DaMod0

Let $\alpha' =_d \alpha^{D}{}_{D'}$, $\beta' =_d \beta^{D}{}_{D'}$, meaning D' is substituted for D in α and β.

Assume that $X \subseteq_d$ Objects and that D, D' : DaMod0.

There are three cases to consider.

C1 Case $D =_d$ EmpDaMod0
Then $DObjs = \varnothing$ so from the definition of Tear we have
Tear(D, X) = EmpDaMod0 $\in$ DaMod0 and so α is true.

C2 Case D AddedEn D'
Then by the definition of AddedEn $DObjs \subseteq D'Objs$ and $D'Objs$ - $DObjs$ is a singleton set whose member is a set of entities. Also $D'Conn = DConn$.

Let $E \subseteq_d$ Entities be such that $\{E\} = D'Objs$ - $DObjs$

For any $t : DObjs$ we have $t \in D'Objs$ and $DAncs(t) = D'Ancs(t)$,
so $\beta' \Rightarrow \beta$. Now consider two cases.

C2.C1 Case $E \notin X$
Then from the definition of Tear we have Tear(D', X) = Tear(D, X). If Tear(D, X) $\in$ DaMod0 then Tear(D', X) $\in$ DaMod0.

C2.C2 Case $E \in X$
Then from the definition of Tear the object E occurs in Tear(D', X) but not in

Tear(D, X). We wish to prove that Tear(D, X) AddedEn Tear(D', X). Then if Tear(D, X) $\in$ DaMod0 we also have Tear(D', X) $\in$ DaMod0.

But from the definition of Tear we have Tear(D', X)*Objs* = Tear(D, X)*Objs* $\cup$ $\{E\}$ and Tear(D', X)*Conn* = Tear(D, X)*Conn*. Also E is a member of *D'Objs* and so obeys the requirements for disjointness, etc. Altogether, we do have Tear(D, X) AddedEn Tear(D', X).

Thus in case C2 Tear(D, X) $\in$ DaMod0 $\Rightarrow$ Tear(D', X) $\in$ DaMod0 and $\beta' \Rightarrow \beta$. We conclude that if α is true then α' is true (as $\neg\,\alpha' \Rightarrow \neg\,\alpha$).

C3 Case D AddedRo D'

Then by the definition of AddedRo *DObjs* $\subseteq$ *D'Objs* and *D'Objs* - *DObjs* is a singleton set whose member is a set of roles disjoint from any member of *DObjs*. Also *DConn* and *D'Conn* agree on all roles occurring in both D and D'.

Let $R \subseteq_d$ Roles be such that $\{R\}$ = *D'Objs* - *DObjs*

For any t : *DObjs* we have $t \in$ *D'Objs* and *DAncs*(t) = *D'Ancs*(t), so $\beta' \Rightarrow \beta$. Now consider two cases.

C3.C1 Case $R \notin X$

Then as in case C2.C1 if Tear(D, X) $\in$ DaMod0 then Tear(D', X) $\in$ DaMod0.

C3.C2 Case $R \in X$

Then as in case C2.C2 we wish to prove that Tear(D, X) AddedRo Tear(D', X), but we need an extra condition here.

As in case C2.C2 the requirements for AddedRo are obeyed except perhaps for the requirement that for each role r : R we have *D'Conn*(r) $\in$ Tear(D, X)*Objs*. This will be obeyed if each immediate predecessor of R is a member of both *DObjs* and of X. As D AddedRo D' we can be sure that the immediate predecessors of R are members of *DObjs*. We need only require that they are also members of X. This will be so if β' is true.

Thus in case C3 [Tear(D, X) $\in$ DaMod0 $\wedge$ β'] $\Rightarrow$ Tear(D', X) $\in$ DaMod0 and $\beta' \Rightarrow \beta$. We conclude that if α is true then α' is true.

Finally, from cases C1, C2, C3 we conclude that α is true for every member of DaMod0 and hence that TearProp3 is true.

□

Diff

Function on members of PreMod : Remove part of a model

Given any P, $P1$: PreMod then

Diff(P, $P1$) $=_d$ P' where P' : PreMod and

$P'Objs =_d PObjs \setminus P1Objs$

$P'ConnGr =_d PConnGr \setminus P1ConnGr$

Diff Properties 1

Some properties of the function Diff

DiffProp3 .:
$\forall D$: DaMod0 $\bullet$ $\forall X \subseteq_d$ Objects $\bullet$
[$\forall t$: ($DObjs \cap X$) $\bullet$ $DDescs(t) \subseteq X$] $\Rightarrow$ Diff(D, Tear(D, X)) $\in$ DaMod0

DiffProp4 .:
$\forall D$: DaMod0 $\bullet$ $\forall t$: $DObjs$ $\bullet$ Diff(D, Tear(D, $DDescIds(t)$)) $\in$ DaMod0

DiffProp4 is an application of DiffProp3 so we need prove only DiffProp3.

To prove
Property DiffProp3 that when a certain precondition is true then Diff transforms a member of DaMod0 into a member of DaMod0. That is, to prove that
$\forall D$: DaMod0 $\bullet$ $\forall X \subseteq_d$ Objects $\bullet$
[$\forall t$: ($DObjs \cap X$) $\bullet$ $DDescs(t) \subseteq X$] $\Rightarrow$ Diff(D, Tear(D, X)) $\in$ DaMod0

The proof is by structural induction on DaMod0 and is like that of TearProp3 with some cases interchanged.

Let $\beta =_{\text{SYM}}$ [$\forall t$: ($DObjs \cap X$) $\bullet$ $DDescs(t) \subseteq X$]

Let $\alpha =_{\text{SYM}}$ β $\Rightarrow$ Diff(D, Tear(D, X)) $\in$ DaMod0

Let $\alpha' =_d \alpha^{D}{}_{D'}$, $\beta' =_d \beta^{D}{}_{D'}$, meaning D' is substituted for D in α and β.

Assume that $X \subseteq_d$ Objects and that D, D' : DaMod0.

There are three cases to consider.

C1 Case $D =_d$ EmpDaMod0
Then $DObjs = \varnothing$ and $DConnDef = \varnothing$ so from the definition of Diff we have Diff(D, Tear(D, X)) = EmpDaMod0 $\in$ DaMod0 and so α is true.

C2 Case D AddedEn D'
Then by the definition of AddedEn $DObjs \subseteq D'Objs$ and $D'Objs$ - $DObjs$ is a singleton set whose member is a set of entities. Also $D'Conn = DConn$.

Let $E \subseteq_d$ Entities be such that $\{E\} = D'Objs - DObjs$

For any t : $DObjs$ we have $t \in D'Objs$ and $DDescs(t) = D'Descs(t)$, since E is not a descendant of t in D', so $\beta' \Rightarrow \beta$. Now consider two cases.

C2.C1 Case $E \in X$
Then from the definition of Tear we have $E \in$ Tear(D', X)$Objs$. Thus from the definition of Diff we have Diff(D', Tear(D', X)) = Diff(D, Tear(D, X)).
If Diff(D, Tear(D, X)) $\in$ DaMod0 then Diff(D', Tear(D', X)) $\in$ DaMod0.

C2.C2 Case $E \notin X$
Then from the definition of Tear we have
$E \notin$ Tear(D', X)$Objs$ = Tear(D, X)$Objs$.

We wish to prove that $\text{Diff}(D, \text{Tear}(D, X))$ AddedEn $\text{Diff}(D', \text{Tear}(D', X))$. Then if $\text{Diff}(D, \text{Tear}(D, X)) \in \text{DaMod0}$ we also have $\text{Diff}(D', \text{Tear}(D', X)) \in \text{DaMod0}$.

But from the definition of Diff we have

$$\text{Diff}(D', \text{Tear}(D', X))Objs = \text{Diff}(D, \text{Tear}(D, X))Objs \cup \{E\}$$

and

$$\text{Diff}(D', \text{Tear}(D', X))Conn = \text{Diff}(D, \text{Tear}(D, X))Conn.$$

Also E is a member of $D'Objs$ and so obeys the requirements for disjointness, etc. Altogether, we do have

$$\text{Diff}(D, \text{Tear}(D, X)) \text{ AddedEn } \text{Diff}(D', \text{Tear}(D', X)).$$

Thus in case C2

$\text{Diff}(D, \text{Tear}(D, X)) \in \text{DaMod0} \;\Rightarrow\; \text{Diff}(D', \text{Tear}(D', X)) \in \text{DaMod0}$

and $\beta' \Rightarrow \beta$. We conclude that if α is true then α' is true.

C3 Case D AddedRo D'

Then by the definition of AddedRo $DObjs \subseteq D'Objs$ and $D'Objs - DObjs$ is a singleton set whose member is a set of roles disjoint from any member of $DObjs$. Also $DConn$ and $D'Conn$ agree on all roles occurring in both D and D'.

Let $R \subseteq_d$ Roles be such that $\{R\} = D'Objs - DObjs$

For any $t : DObjs$ we have $t \in D'Objs$ and $DDescs(t) = D'Descs(t) \setminus \{R\}$, so $\beta' \Rightarrow \beta$. Now consider two cases.

C3.C1 Case $R \in X$

Then as in case C2.C1 if $\text{Diff}(D, \text{Tear}(D, X)) \in \text{DaMod0}$ then $\text{Diff}(D', \text{Tear}(D', X)) \in \text{DaMod0}$.

C3.C2 Case $R \notin X$

Then as in case C2.C2 we wish to prove that

$$\text{Diff}(D, \text{Tear}(D, X)) \text{ AddedRo } \text{Diff}(D', \text{Tear}(D', X)),$$

but we need an extra condition here.

As in case C2.C2 the requirements for AddedRo are obeyed except perhaps for the requirement that for each role $r : R$ we have $D'Conn(r) \in \text{Diff}(D, \text{Tear}(D, X))Objs$. This will be obeyed if each immediate predecessor of R is a member of $DObjs$ but not of X. As D AddedRo D' we can be sure that the immediate predecessors of R are members of $DObjs$. We need only require that none are members of X. This will be so if R is not an immediate successor of any member of $(D'Objs \cap X)$. This in turn will be so if any immediate successor of a member of $(D'Objs \cap X)$ is itself a member of X. (Remember $R \notin X$). That is, if β' is true.

Thus in case C3

$[\text{Diff}(D, \text{Tear}(D, X)) \in \text{DaMod0} \wedge \beta'] \;\Rightarrow\; \text{Diff}(D', \text{Tear}(D', X)) \in \text{DaMod0}$

and $\beta' \Rightarrow \beta$. We conclude that if α is true then α' is true.

Finally, from cases C1, C2, C3 we conclude that α is true for every member of DaMod0 and hence that DiffProp3 is true.

□

<u>Merge</u>

Function on members of PreMod : Merge two models together

Given any $P1, P2$: PreMod with

(Pre 1) $\forall r : (P1ConnDef \cap P2ConnDef) \bullet P1Conn(r) = P2Conn(r)$

then

Merge$(P1, P2) =_d P'$ where P' : PreMod and

$P'Objs =_d P1Objs \cup P2Objs$

$P'ConnGr =_d P1ConnGr \cup P2ConnGr$

<u>Merge   Properties 1</u>

Some properties of the function Merge

MergeProp3 .:
$\forall P$: PreMod $\bullet$ $\forall X \subseteq_d$ Objects $\bullet$
Merge(Tear(P, X), Diff(P, Tear(P, X))) = P

MergeProp4 .:
$\forall D1, D2$: DaMod0 $\bullet$
$[\ (\forall x, y : (D1Objs \cup D2Objs) \bullet x = y \vee x \cap y = \varnothing) \wedge$
$(\forall r : (D1Roles \cap D2Roles) \bullet D1Conn(r) = D2Conn(r))\ ]$
$\Rightarrow$
Merge$(D1, D2) \in$ DaMod0

MergeProp3 is a straightforward application of the identity $(A \cap X) \cup (A \setminus X) = A$. We need prove only MergeProp4.

To prove
Property MergeProp4 that when a certain precondition is true then Merge transforms two members of DaMod0 into a member of DaMod0. That is, to prove that

$\forall D1, D2$: DaMod0 $\bullet$
$[\ (\forall x, y : (D1Objs \cup D2Objs) \bullet x = y \vee x \cap y = \varnothing) \wedge$
$(\forall r : (D1Roles \cap D2Roles) \bullet D1Conn(r) = D2Conn(r))\ ]$
$\Rightarrow$
Merge$(D1, D2) \in$ DaMod0

The proof is by structural induction on DaMod0, using the variable $D1$, and is like that of TearProp3 with some cases interchanged.

Let $\beta =_{SYM}$
$[\ (\forall x, y : (D1Objs \cup D2Objs) \bullet x = y \vee x \cap y = \varnothing) \wedge$
$(\forall r : (D1Roles \cap D2Roles) \bullet D1Conn(r) = D2Conn(r))\ ]$

Let $\alpha =_{SYM} \beta \Rightarrow$ Merge$(D1, D2) \in$ DaMod0

Let $\alpha' =_d \alpha^{D1}{}_{D1'}$, $\beta' =_d \beta^{D1}{}_{D1'}$, meaning *D1'* is substituted for *D1* in α and β.

Assume that *D1*, *D1'*, *D2* : DaMod0. Observe that whenever β is true then the precondition MergePre1 is satisfied for (*D1*, *D2*), and whenever β' is true then MergePre1 is satisfied for (*D1'*, *D2*).

There are three cases to consider.

C1 Case $D1 =_d$ EmpDaMod0

Then *D1Objs* = ∅, *D1ConnGr* = ∅, and MergePre1 is satisfied vacuously for (*D1*, *D2*). From the definition of Merge we have Merge(*D1*, *D2*) = *D2* ∈ DaMod0 so α is true.

C2 Case *D1* AddedEn *D1'*

Then by the definition of AddedEn *D1Objs* ⊆ *D1'Objs* and *D1'Objs* - *D1Objs* is a singleton set whose member is a set of entities. Also *D1'Conn* = *D1Conn*.

Let $E \subseteq_d$ Entities be such that {*E*} = *D1'Objs* - *D1Objs*

Clearly β' ⇒ β. Assume that β' is true and that Merge(*D1*, *D2*) ∈ DaMod0. Now consider two cases.

C2.C1 Case *E* ∈ *D2Objs*

Then from the definition of Merge we have Merge(*D1'*, *D2*) = Merge(*D1*, *D2*) so Merge(*D1'*, *D2*) ∈ DaMod0.

C2.C2 Case *E* ∉ *D2Objs*

Then from the definition of Merge the object *E* occurs in Merge(*D1'*, *D2*) but not in Merge(*D1*, *D2*). We wish to prove that Merge(*D1*, *D2*) AddedEn Merge(*D1'*, *D2*). Then we have Merge(*D1'*, *D2*) ∈ DaMod0.

But from the definition of Merge we have

Merge(*D1'*, *D2*)*Objs* = Merge(*D1*, *D2*)*Objs* ∪ {*E*} and
Merge(*D1'*, *D2*)*Conn* = Merge(*D1*, *D2*)*Conn*.

Also *E* is a member of *D1'Objs* and so obeys the requirements for being non-empty, etc; from β' it obeys the requirements for disjointness. Altogether, we do have Merge(*D1*, *D2*) AddedEn Merge(*D1'*, *D2*).

Thus in case C2

[Merge(*D1*, *D2*) ∈ DaMod0 ∧ β'] ⇒ Merge(*D1'*, *D2*) ∈ DaMod0
and β' ⇒ β. We conclude that if α is true then α' is true.

C3 Case *D1* AddedRo *D1'*

Then by the definition of AddedRo *D1Objs* ⊆ *D1'Objs* and *D1'Objs* - *D1Objs* is a singleton set whose member is a set of roles disjoint from any member of *D1Objs*. Also *D1Conn* and *D1'Conn* agree on all roles occurring in both *D1* and *D1'*.

Let $R \subseteq_d$ Roles be such that {*R*} = *D1'Objs* - *D1Objs*

Clearly β' ⇒ β. Assume that β' is true and that Merge(*D1*, *D2*) ∈ DaMod0. Now consider two cases.

C3.C1 Case *R* ∈ *D2Objs*

Then from β' we have for all *r* : *R* that *D1'Conn*(*r*) = *D2Conn*(*r*). From the definition of Merge we have Merge($D1'$, $D2$) = Merge($D1$, $D2$) so Merge($D1'$, $D2$) $\in$ DaMod0.

C3.C2 Case $R \notin D2Objs$

Then as in case C2.C2 we wish to prove that Merge($D1$, $D2$) AddedRo Merge($D1'$, $D2$).

As in case C2.C2 the requirements for AddedRo are obeyed except that we must confirm for each role $r : R$ that $D1'Conn(r) \in$ Merge($D1$, $D2$)$Objs$. But $D1$ AddedRo $D1'$ so we can be sure that $D1'Conn(r) \in D1Objs \subseteq$ Merge($D1$, $D2$)$Objs$.

Thus in case C3

[Merge($D1$, $D2$) $\in$ DaMod0 $\wedge$ β'] $\Rightarrow$ Merge($D1'$, $D2$) $\in$ DaMod0
and $\beta' \Rightarrow \beta$. We conclude that if α is true then α' is true.

Finally, from cases C1, C2, C3 we conclude that α is true for every $D1$: DaMod0 and hence that MergeProp3 is true.

□

<u>Move</u>

Function on members of PreMod : Change a role association (or introduce a new one)

Given any P : PreMod, r : Roles, x : Objects then

Move(P, r, x) $=_d$ P' where P' : PreMod and

$P'Objs =_d PObjs$

$P'Conn =_d (PConn \mid r \mapsto x)$ Make $P'Conn(r) = x$

<u>Move Properties 1</u>

Some properties of the function Move

MoveProp4 .:
$\forall P$: PreMod $\bullet$ $\forall r, r'$: Roles $\bullet$ $\forall x, x'$: Objects $\bullet$
$r' \neq r \Rightarrow$ Move(Move(P, r, x), r', x') = Move(Move(P, r', x'), r, x)

MoveProp5 .:
$\forall D$: DaMod0 $\bullet$ $\forall r : DRoles$ $\bullet$ $\forall x : DObjs$ $\bullet$
$x \notin DDescIds(DRo(r)) \Rightarrow$ Move(D, r, x) $\in$ DaMod0

MoveProp4 is immediate from the definition of Move. We need prove only MoveProp5.

To prove
Property MoveProp5 that when a certain precondition is true then Move transforms a member of DaMod0 into a member of DaMod0. That is, to prove that

$$\forall D : \text{DaMod0} \bullet \forall r : DRoles \bullet \forall x : DObjs \bullet$$
$$x \notin DDescIds(DRo(r)) \Rightarrow \text{Move}(D, r, x) \in \text{DaMod0}$$

Recall that for any D : DaMod0 we have $DRoles =_{\mathrm{d}} \bigcup \{ R : DObjs \mid R \subseteq \text{Roles} \}$, the roles occurring in D. We will dispose of the vacuous case first, them move on to the bulk of the proof.

Assume that D : DaMod0.

C1 Case $DRoles = \varnothing$
Then
$\forall r : DRoles \bullet \forall x : DObjs \bullet$
$x \notin DDescIds(DRo(r)) \Rightarrow \text{Move}(D, r, x) \in \text{DaMod0}$
is true vacuously. Equivalently, one of the preconditions, $r \in DRoles$, for applying Move is false.

C2 Case $DRoles \neq \varnothing$
Then $DObjs \neq \varnothing$.

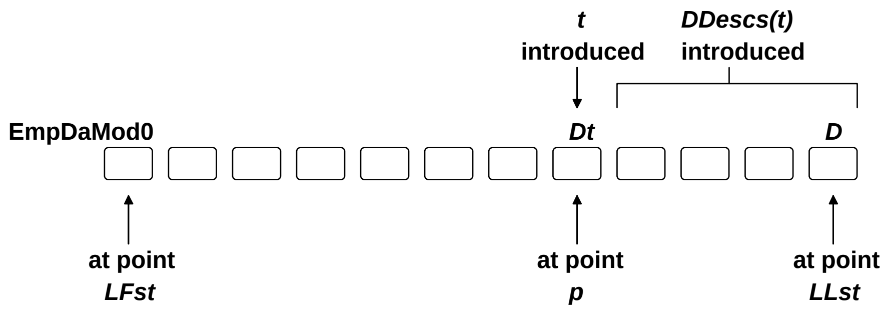
Assume that $r : DRoles$ and $x : DObjs$. Then r is a member of the object $DRo(r) \in DObjs$.

Let $t =_{\mathrm{d}} DRo(r)$, so $r \in t \in DObjs$.

The proof uses the construction sequences defined in Section 4.5.4. Recall that a construction sequence for D is a list of members of DaMod0, starting with EmpDaMod0 and ending with D. Each successive element of the list introduces one more member of *DObjs*. In the proof accompanying the definition of the function CompSeq we showed that successive elements are related either by AddedEn or by AddedRo.

In outline, we use a specially chosen construction sequence illustrated in Figure 5.2.2.1 below. The order in which objects are introduced gives us a construction sequence where the members of $DDescIds(t)$ are introduced last of all.

Figure 5.2.2.1 A special construction sequence for *D*



We then form a copy of the list where some elements are changed. Elements prior to the introduction of t are unchanged (those to the left of Dt in the figure). The rest are changed to show the result of applying Move to reassign the role r to the object x. We then prove that successive elements of this altered list are also related either by

AddedEn or by AddedRo, thus proving that Move(D, r, x) $\in$ DaMod0. At the first element in the sequence where r is reassigned we can reassign it to any member of $DObjs$ - $DDescIds(t)$, which is what we wanted to prove.

As it happens, it is more convenient to use completion sequences in the proof. We use a sequence that starts at the element whose objects are $DObjs$ - $DDescIds(t)$. In the figure this is the element immediately to the left of Dt.

Now we continue case C2. Recall that the class ListV of lists, the set ObChos of choice functions, and the function CompSeq that returns a completion sequence are defined in Section 4.5.4.

First we build a suitable completion sequence. Form the starting sub-model $Ds =_d$ Diff(D, Tear(D, $DDescIds(t)$)). From the definition of Diff and Tear we can be sure that $DsObjs = DObjs$ - $DDescIds(t)$. We can also be sure that $DsObjs \subseteq DObjs$, and $DsConnGr \subseteq DConnGr$, and by DiffProp4 $Ds \in$ DaMod0, so (Ds, D) meets the preconditions for the function CompSeq.

Pick any choice function C : ObChos. The completion sequence we will use is the list L : ListV such that $L =_d$ CompSeq$_C$(Ds, D). Recall that each element of L is a member of DaMod0. The first element, $LV(LFst)$, of L is Ds. The second, $Dt =_d LV(LS(LFst))$, is the one that introduces t. For if not, Dt introduces a descendant of t, so we would have an object of $DtObjs$ with an ancestor, t, that is not a member of $DtObjs$. Any further elements of L introduce the members of $DDescs(t)$. The last element, of course, is D.

Next we build the list L' : ListV of altered models. The primary variable features of L' are

$L'Pts =_d LPts$, $L'S =_d LS$, $LVals =_d$ PreMod, and for $L'V$ we require

```
∀i : L'Pts •
    if     r ∈ LV(i)Roles
    then   L'V(i) =_d Move(LV(i), r, x)
    else   L'V(i) =_d LV(i).
```

Now we wish to prove that each element of L' is a member of DaMod0. The proof is by induction on $L'Pts$, with two cases to consider. Assume that i : $L'Pts$.

C2.C1 Case $i = L'Fst = LFst$

$L'V(L'Fst) = Ds$ by construction as $t \notin DsObjs$ so $r \notin DsRoles$. Ds is a member of DaMod0.

C2.C2 Case $i \in L'SDef = LSDef$

Let $D' =_d L'V(L'S(i))$. Now D' introduces an object t' that is either t or a descendant of t, so t' can only be a set of roles. We wish to prove that $L'V(i)$ AddedRo D' and hence that $L'V(i) \in$ DaMod0 $\Rightarrow D' \in$ DaMod0. Note that in L we have $LV(i)$ AddedRo $LV(LS(i))$, and that $L'V(i)Objs = LV(i)Objs$ and $L'V(L'S(i))Objs = LV(LS(i))Objs$.

In the list L we know that t' is introduced by $LV(LS(i))$ so t' and its role associations in D meet the requirements for $LV(i)$ AddedRo $LV(LS(i))$. Consequently t' and its role associations in D' meet the requirements for $L'V(i)$ AddedRo D' except perhaps if $r \in t'$. If $r \in t'$ then $t' = t$, D' is the second element, Dt, of the list, and $L'V(i)$ is Ds. Then t' will meet the requirements only if $DtConn(r) \in DsObjs = DObjs$ - $DDescIds(t)$. In other words, for any $i : L'SDef$ we have

$$[L'V(i) \in \mathsf{DaMod0} \ \wedge \ x \notin DDescIds(t)] \ \Rightarrow \ L'V(L'S(i)) \in \mathsf{DaMod0}.$$

From case C2.C1 and C2.C2 we conclude that if $x \notin DDescIds(t)$ then

$\mathsf{Move}(D, r, x) = L'V(LLst) \in \mathsf{DaMod0}$.

Finally, from case C1 and C2 we conclude that MoveProp5 is true.

□

5.3 Fact Type structure

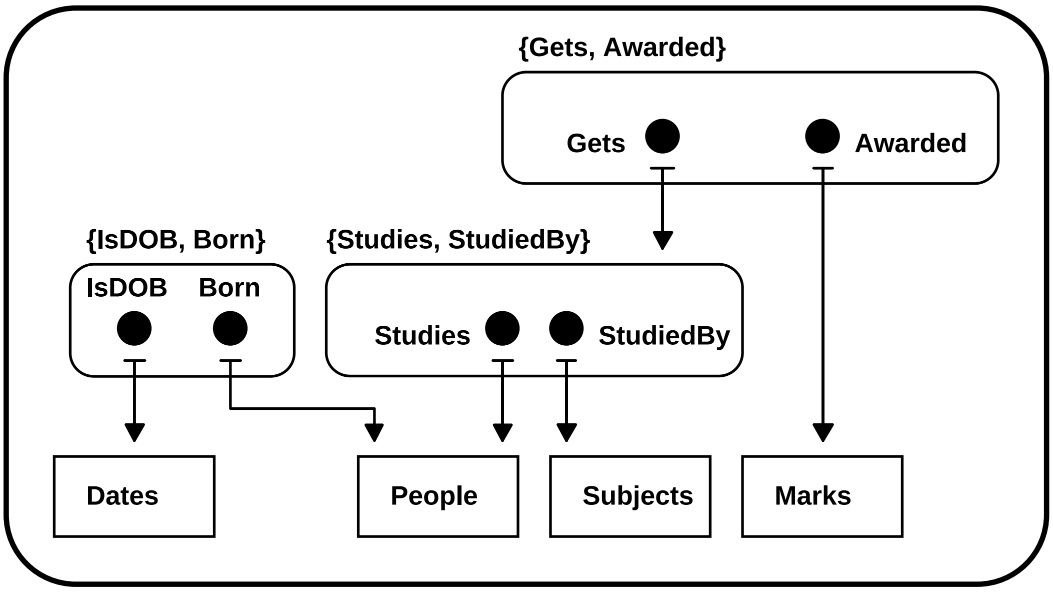
Suppose we say that this Fact Type in this data model is the same as that Fact Type in that data model. What might we mean? After all, we have found that a Fact Type symbol really describes a set of roles. Only if the data model is well formed can we be sure that the symbol identifies a cartesian product, alias Fact Type. Do we mean that we have the same set of roles, or the same cartesian product, or what? We need to be clear what we mean when we use the word "same".

This is the first of three sections which give meanings to the word "same". Two different meanings will be given in Sections 5.4 and 5.5.

5.3.1 Outline

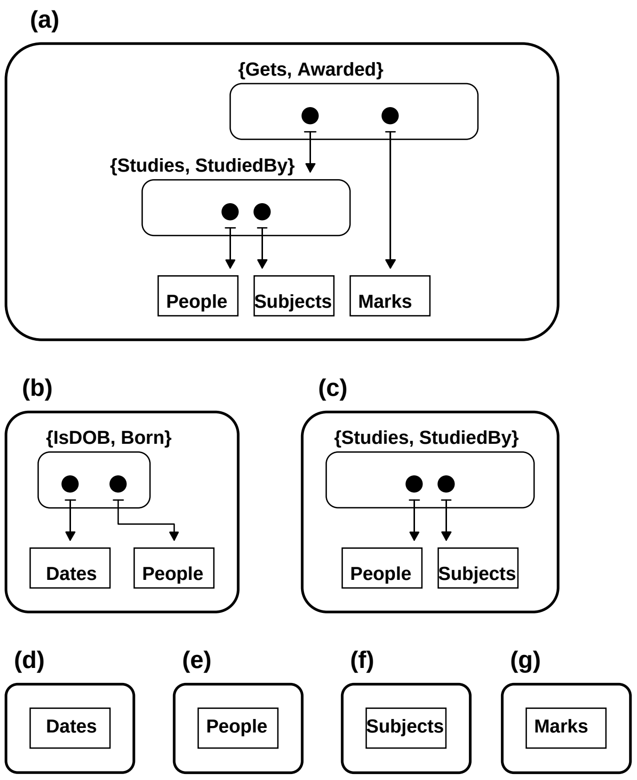
Consider the member Dx of DaMod0 given in Figure 5.3.1.1. It has seven objects. If we take each object in isolation we have three sets of roles and four sets of entities. None of these tells us much about Dx.

Figure 5.3.1.1 A member of DaMod0



However, if we tear out each object along with its ancestors we know from the discussion of the Tear function in the previous section that we will obtain members of DaMod0, the seven pictured in Figure 5.3.1.2 below. These seven do tell us something about Dx.

Figure 5.3.1.2 A description of each object



We know from the discussion of ancestors in Section 4.4 that the cartesian products identified by the objects in Figure 5.3.1.2 are the same as those in Dx. (For the Entity Types it is the nullary cartesian product, by decree. Nevertheless, it is the same.) Thus we can say that the Fact Type, alias cartesian product, identified by {Gets, Awarded} in Figure 5.3.1.2(a) is one of the Fact Types of Dx, and we can say the same about {IsDOB, Born} and {Studies, StudiedBy} in (b) and (c).

We will say that Figure 5.3.1.2(a) describes the **type structure** of {Gets, Awarded} in Dx, and we will define a secondary feature that gives us the type structure of any object. For any model *D* : DaMod0 and object *t* : *DObjs* then *DTyStruc*(*t*) is the member of DaMod0 that is *t*'s type structure in *D*. *DTyStruc*(*t*) is Tear(*D*, *DAncIds*(*t*)). We can collect all possible type structures together to give us a subset of DaMod0 which we will call TypeStruc.

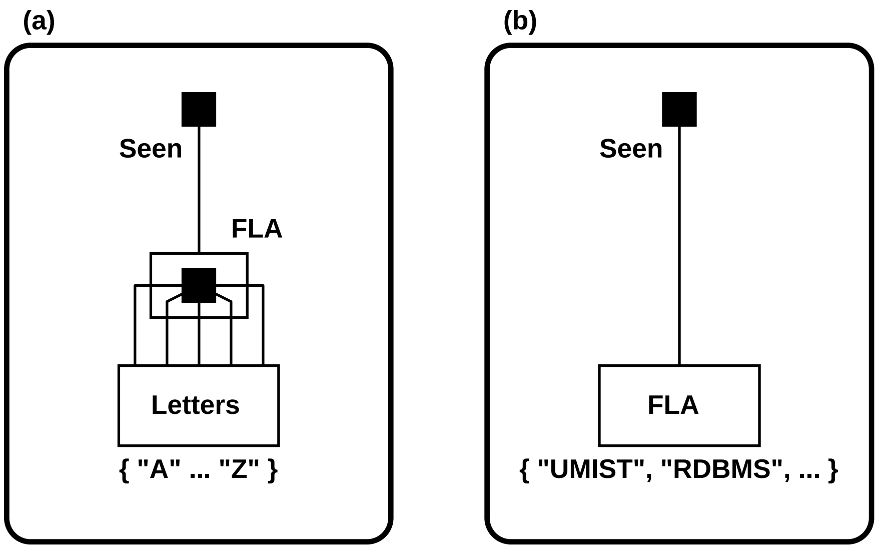
Notice that we could use Merge operations to merge Figures 5.3.1.2(a)…(g) together to reconstruct Dx. In general, any member of DaMod0 can be built by merging members of TypeStruc together (with due regard to preconditions).

Now we can give one particular meaning to the word "same". Two objects occurring in two data models are the same in this sense iff they have the same type structure. If they are the same then they both identify the same cartesian product (trivially in the case of Entity Types). In our model of data models this kind of sameness is expressed by an equivalence relation, StrucEq.

Sameness has implications for copying. If we copy something we expect that the copy will be the same as the original in some sense. Here, we can conclude that an editor should not copy objects from one data model to another; it should copy type structures.

Note that an object can identify the same Fact Type in different data models without having the same type structure. Suppose we wish to record all the five-letter acronyms that we have seen. The data model in Figure 5.3.1.3(a) below specifies a database to do this. There the role Seen is associated with the cartesian product FLA consisting of all 5-tuples of letters. However, we know that most database products will implement five-letter strings as a primitive type so we can specify the database with the data model in Figure 5.3.1.3(b). There Seen is associated with an Entity Type consisting of all five-letter strings. But this set of strings could be FLA, in which case the object {Seen} identifies the same cartesian product in both (a) and (b), but has a different type structure in each.

Figure 5.3.1.3 Two type structures, one cartesian product



This does not mean that the definition of sameness is deficient. Different type structures imply different implementation requirements. For instance, in Figure 5.3.1.3 listing all recorded acronyms that end with "Z" would require very different queries in the two databases.

Note that type structure does not tell us everything about an object. In particular, it says nothing about the object's descendants.

To sum up, we have given one precise meaning to statements of the form "This Fact Type is the 'same' as that Fact Type". We have also seen that a NIAM conceptual data model can be described as a collection of type structures. Perhaps this is the thinking behind the use of "predicators" in the Predicator Model (Section 2.1.2).

5.3.2 Details

This section provides definitions and properties concerning type structure.

First we define the functions that give us the type structure of each object of each member of DaMod0 and follow it with the property that justifies the definition.

D : DaMod0 Secondary features 5

A secondary feature of members of DaMod0 : Type structure

DTyStruc : *DObjs* $\rightarrow$ DaMod0 Given *t* : *DObjs* then *DTyStruc*(*t*) is *t*'s type structure in *D*

DDefTyStruc .: $\forall t$: *DObjs* $\bullet$ *DTyStruc*(*t*) $=_d$ Tear(*D*, *DAncIds*(*t*))

D : DaMod0 Properties 5

A property of each member of DaMod0

DProp5.3 .: $\forall t$: *DObjs* $\bullet$ *DTyStruc*(*t*)*Cart*(*t*) = *DCart*(*t*)

Recall that for any *D* : DaMod0 and *t* : *DObjs* we can be sure that Tear(*D*, *DAncIds*(*t*)) $\in$ DaMod0. Thus *DTyStruc*(*t*) is well defined as a member of DaMod0. Property *DProp5.3* is immediate from the definition of Tear and the preservation property of *DCart* (*DProp4.14* in Section 4.4.2).

Next we collect together all the members of DaMod0 that describe the type structure of some object.

TypeStruc $\subseteq_d$ DaMod0

Those members of DaMod0 that describe a type structure (with two alternative definitions)

TypeStruc $=_d$ { *DTyStruc*(*t*) | *D* : DaMod0 $\wedge$ *t* : *DObjs* }

= { *D* : DaMod0 | $\exists t$: *DObjs* $\bullet$ *DObjs* = *DAncIds*(*t*) }

That the two definitions are equivalent is immediate from the definitions of Tear and *DTyStruc*.

Finally we define the equivalence relation StrucEq and state an immediate property.

<u>StrucEq</u>

Predicate that is true of two objects of two members of DaMod0 iff they have the same type structure

Given any *D1*, *D2* : DaMod0, *t1* : *D1Objs*, *t2* : *D2Objs* then

StrucEq((*D1*, *t1*), (*D2*, *t2*)) $\Leftrightarrow_d$ *D1TyStruc*(*t1*) = *D2TyStruc*(*t2*)

<u>StrucEq   Properties 1</u>

A property of StrucEq

StrucEqProp1 .:
$\forall$*D1*, *D2* : DaMod0, *t1* : *D1Objs*, *t2* : *D2Objs* •
StrucEq((*D1*, *t1*), (*D2*, *t2*)) $\Rightarrow$ *D1Cart*(*t1*) = *D2Cart*(*t2*)

The property is immediate from *DProp5.3* and the definition of StrucEq. Note that the reverse implication cannot be proved. A proof would require that no subset of Entities is a cartesian product defined by some member of DaMod0; we have not required Entities to have this property.

5.4 Simple equivalences

Suppose we wish to record facts about traffic lights. We might construct a conceptual data model that uses the role IsLit and the Entity Type {Red, Yellow, Green}. What if we use the role IsOn and the Entity Type {Scarlet, Orange, Viridian} instead? Would we have the same data model and the same database?

This section gives a second meaning to the word "same". We will define an isomorphism, alias structure-preserving transformation, on the members of PreMod and we will prove that it has the appropriate properties.

5.4.1 Outline

A change from IsOn to IsLit or from Red to Scarlet could be described as a change of name rather than as a change of the things the names refer to. However, roles are placeholders. It should be possible to change them without causing problems. As the final choice of Entity Types can be left until implementation it should be possible to change entities as well without causing problems.

In the model of core data models the members of the sets Roles and Entities are modelling elements. We should certainly be able to select any member of Roles to model IsOn and any member of Entities to model Red.

We will define a function that describes changes of roles and entities and we will confirm that a changed data model is the "same" as the original for many purposes. We will ensure that the data model's structure is preserved : Entity Types are translated into Entity Types and index sets into index sets; each association of a role with a domain is translated into the association of the replacement role with the replacement domain.

In the model of data models we will call the function BaseConv, for **base conversion**. On being given a member of PreMod and a conversion rule it returns the member of PreMod that incorporates the changes, with the necessary preservation of structure.

In the traffic light example we might specify the following conversion rule :

Red	$\mapsto$	Scarlet
Yellow	$\mapsto$	Orange
Green	$\mapsto$	Viridian
IsLit	$\mapsto$	IsOn

with any other roles and entities unchanged. Notice that there should be some restrictions on the conversion rule. If the entities Red and Yellow both change to Scarlet then the result may be a member of PreMod but not one capable of describing information about traffic lights. If the role IsLit changes to the entity Orange then the result will not be a member of PreMod. If the rule says nothing about the entity Green then we do not know what the result will be.

Thus, if the member *P* of PreMod is to be changed according to the conversion rule *B* we will require that *B* is a one-to-one correspondence (a bijection) that is defined for each role and entity mentioned in *P*, and that these roles must map to roles and these entities to entities. Then the result, BaseConv(*P*, *B*), will be the changed member of PreMod.

It is tempting to require *B* to be defined for all members of Roles and Entities. Unfortunately, if *B* maps Entities into a proper subset of Entities then the inverse conversion rule B^{-1} would not obey the requirements. We will require only that *B* is defined for the minimum necessary members of Roles and Entities.

BaseConv has three significant properties. The first property is an immediate consequence of the preconditions. If the conversion rule *B* meets the requirements for converting *P* and converts it into *P'* then the inverse rule B^{-1} obeys the requirements and converts *P'* back into *P*. Base conversion of a data model can be undone.

The second property is that if any member of DaMod0 is converted using a conversion rule that meets the requirements then the result is also a member of DaMod0. Base conversion of a well-formed data model gives us a well-formed data model. Note that the conversion rule is single rule common to all the Entity Types of the data model. If there were a separate rule for each Entity Type then the result could be an ill-formed data model with overlapping Entity Types.

The third property concerns tuples. Remember that the primary job of a core data model is to define a set of tuples, or, equivalently, a set of cartesian products. In the traffic light example we might have the information item
　　'Red is lit at 12:30'
represented by the tuple
　　(IsLit $\mapsto$ Red, At $\mapsto$ 12:30).
We would naturally expect that the change of roles and entities given earlier would translate this tuple into the tuple
　　(IsOn $\mapsto$ Scarlet, At $\mapsto$ 12:30)
representing the information item
　　'Scarlet is on at 12:30'.

The third property is that if *D* : DaMod0 is converted to *D'* : DaMod0 by a rule that obeys the requirements then there is a one-to-one correspondence between the set of tuples defined by *D* and the set defined by *D'*. Moreover, one of the typically many correspondences is "natural" in the way illustrated above. This natural correspondence continues when the tuples are themselves elements of other tuples, though not as easy to describe. (We will see a way to describe it in Section 5.5).

There are two uses for base conversion. The first is theoretical. The properties justify our lack of concern for precisely which members of Roles and Entities we use to model a given data model. They also justify the use of data modelling notations that assert the existence of distinct roles without stating explicitly which roles they are.

The second use is practical. A common occurrence is that a data model is developed as several different models which are then merged together in a view integration process. Models that are to be merged must obey the preconditions for merging. Base conversion can be used to make different things the same where, for instance, Red has been used as a traffic light colour in one data model and Scarlet in another. And it can be used to make the same things different where, for instance, Red has been used as a traffic light colour in one data model and a project status code in another.

Now we can give another meaning to the word "same". Two data models are the same in this sense iff there is a rule that obeys the requirements for conversion and it converts one into the other. In our model of data models this kind of sameness is expressed by a second equivalence relation BaseEq.

Remember that a core data model says nothing about the business rules that state when and how the database is to be updated. It is possible for two data models to be the "same" in the BaseEq sense even though the databases are going to be used for very different purposes.

5.4.2 Details

In this section we define base conversion and the equivalence relation derived from it. We also state and prove some properties.

The image operator is used in many places. The definition, adapted from Enderton [1977], p44, used here is

---

[[ ]] (Image operator)

Operator that returns the image of a set under a function

Given any F : Function, X : Set then

$F[[\ X\ ]] =_d \{\ F(x) \mid x : (X \cap FDef)\ \}$

We define base conversion as the structure-preserving replacement of the roles and entities of a given member of PreMod. The precondition has been chosen to ensure that the result is a well-defined member of PreMod, with distinct roles transformed into distinct roles, distinct entities transformed into distinct entities, and the conversion always invertible. Notice that in data modelling we assume that if entities of different types are distinct then the distinction is significant.

BaseConv

Function that replaces each role and entity mentioned in a member of PreMod

Given any P : PreMod, B : Bijection with

(Pre 1) All roles mentioned in P map to roles and entities to entities
$\forall X : (PObjs \cup \{PConnDef\} \cup PConnRan) \bullet$
$X \subseteq BDom \wedge$
$(X \subseteq \text{Roles}) \Rightarrow B[\![X]\!] \subseteq \text{Roles}) \wedge$
$(X \subseteq \text{Entities}) \Rightarrow B[\![X]\!] \subseteq \text{Entities})$

then

BaseConv(P, B) $=_\text{d}$ P' where P' : PreMod and

$P'Objs =_\text{d} \{ B[\![t]\!] \mid t : PObjs \}$

$P'ConnDef =_\text{d} B[\![PConnDef]\!]$

$\forall r : PConnDef \bullet P'Conn(B(r)) =_\text{d} B[\![PConn(r)]\!]$

Now we state and prove the first two results : that base conversion is invertible and that DaMod0 is closed under base conversion. The proof of the latter uses results concerning the generators of DaMod0; they are stated and proved separately.

BaseConv Properties 1

Some properties of the function BaseConv

$\forall P$: PreMod $\bullet$ $\forall B$: Bijection $\bullet$
[$\forall X : (PObjs \cup \{PConnDef\} \cup PConnRan) \bullet X \subseteq BDom \wedge$
$(X \subseteq \text{Roles}) \Rightarrow B[\![X]\!] \subseteq \text{Roles}) \wedge (X \subseteq \text{Entities}) \Rightarrow B[\![X]\!] \subseteq \text{Entities})$]
$\Rightarrow$

BaseConvProp1.1 .: Base conversion is invertible
$\forall P'$: PreMod $\bullet$ $P' =_\text{d}$ BaseConv(P, B) $\Rightarrow$
[($\forall X : (P'Objs \cup \{P'ConnDef\} \cup P'ConnRan) \bullet X \subseteq B^{-1}Dom \wedge$
$(X \subseteq \text{Roles}) \Rightarrow B^{-1}[\![X]\!] \subseteq \text{Roles}) \wedge$
$(X \subseteq \text{Entities}) \Rightarrow B^{-1}[\![X]\!] \subseteq \text{Entities})$) $\wedge$
BaseConv(P', B^{-1}) = P]

BaseConvProp1.2 .: BaseConv preserves EmpDaMod0
  BaseConv(EmpDaMod0, *B*) = EmpDaMod0

BaseConvProp1.3 .: BaseConv preserves AddedEn
  $\forall P''$: PreMod $\bullet$
    P'' AddedEn *P* $\Rightarrow$ BaseConv(*P''*, *B*) AddedEn BaseConv(*P*, *B*)

BaseConvProp1.4 .: BaseConv preserves AddedRo
  $\forall P''$: PreMod $\bullet$
    [$\forall X$: *P''ConnRan* $\bullet$ $X \subseteq BDom$ $\wedge$
        ($X \subseteq$ Roles) $\Rightarrow$ *B*[[*X*]] $\subseteq$ Roles) $\wedge$
        ($X \subseteq$ Entities) $\Rightarrow$ *B*[[*X*]] $\subseteq$ Entities)] $\wedge$
    P'' AddedRo *P*
    $\Rightarrow$
    BaseConv(*P''*, *B*) AddedRo BaseConv(*P*, *B*)

BaseConvProp1.5 .: BaseConv preserves DaMod0
  P $\in$ DaMod0 $\Leftrightarrow$ BaseConv(*P*, *B*) $\in$ DaMod0

Note that the condition in BaseConvProp1.4 is necessary : AddedRo allows *P''ConnDef* to overlap the added set of roles. There is no overlap if *P''* $\in$ DaMod0.

Property BaseConvProp1.1 is immediate from the definition of BaseConv and the properties of bijections. We need prove only the others.

To prove
Property BaseConvProp1.2 that BaseConv preserves EmpDaMod0.

Recall that EmpDaMod0Objs = $\varnothing$ = EmpDaMod0ConnDef. Any bijection *B* obeys the preconditions for the conversion of EmpDaMod0. It is now immediate from the definition of BaseConv that BaseConv(EmpDaMod0, *B*) = EmpDaMod0.

□

To prove
Property BaseConvProp1.3 that BaseConv preserves AddedEn.

Assume that *P''*, *P* : PreMod, that *P''* AddedEn *P*, and that *B* : Bijection obeys the precondition for the conversion of *P*.

By the definition of AddedEn *P''Objs* $\subseteq$ *PObjs*, *PObjs* - *P''Objs* $=_d$ {*E*} where $E \subseteq_d$ Entities, and *P''Conn* = *PConn*. Also, *E* is non-empty and disjoint from each member of *P''Objs*. *B* obeys the precondition for the conversion of *P''* as well as of *P*.

From the precondition, for any *t* : *PObjs* we have $t \subseteq BDom$, and so from the definition of BaseConv and the properties of bijections we have
B[[*E*]] $\subseteq$ Entities,
B[[*E*]] $\neq \varnothing$,
B[[*E*]] is disjoint from each member of BaseConv(*P''*, *B*)*Objs*,
BaseConv(*P*, *B*)*Objs* = BaseConv(*P''*, *B*)*Objs* $\cup$ {*B*[[*E*]]},
BaseConv(*P*, *B*)*Conn* = BaseConv(*P''*, *B*)*Conn*.

These are the conditions for BaseConv(P'', B) AddedEn BaseConv(P, B). We conclude that BaseConv preserves AddedEn.

□

To prove
Property BaseConvProp1.4 that BaseConv preserves AddedRo.

Assume that P'', P : PreMod and that P'' AddedRo P.

Assume that B : Bijection obeys the precondition for the conversion of P and that
$\forall X : P''ConnRan \bullet X \subseteq BDom \wedge$
$(X \subseteq \text{Roles}) \Rightarrow B[[\ X\ ]] \subseteq \text{Roles}) \wedge$
$(X \subseteq \text{Entities}) \Rightarrow B[[\ X\ ]] \subseteq \text{Entities})$

By the definition of AddedRo $P''Objs \subseteq PObjs$, $PObjs - P''Objs =_d \{R\}$
where $R \subseteq_d$ Roles, and R is non-empty, finite, and disjoint from each member of $P''Objs$.

In addition, we have $PConnDef = P''ConnDef \cup R$ so $P''ConnDef \subseteq PConnDef$. We are given that B obeys the $P''ConnRan$ part of the precondition. Altogether, B obeys the precondition for the conversion of P'' as well as of P.

For the same reasons as in the AddedEn case we have
$B[[\ R\ ]] \subseteq$ Roles,
$B[[\ R\ ]] \neq \varnothing$,
$B[[\ R\ ]]$ is finite,
$B[[\ R\ ]]$ is disjoint from each member of BaseConv(P'', B)$Objs$,
BaseConv(P, B)$Objs$ = BaseConv(P'', B)$Objs \cup \{B[[\ R\ ]]\}$.

We have $R \subseteq PConnDef$. By the definition of AddedRo, for any $r : PConnDef$, $PConn(r) \in P''Objs$ if $r \in R$, and $PConn(r) = P''Conn(r)$ otherwise. Thus we also have

$\forall r : B[[\ R\ ]] \bullet$
$r \in$ BaseConv(P, B)$ConnDef \wedge$
BaseConv(P, B)$Conn(r) \in$ BaseConv(P'', B)$Objs$,
and
$\forall r$: (BaseConv(P, B)$ConnDef - B[[\ R\ ]]$) $\bullet$
BaseConv(P, B)$Conn(r)$ = BaseConv(P'', B)$Conn(r)$.

These are the conditions for BaseConv(P'', B) AddedRo BaseConv(P, B). We conclude that BaseConv preserves AddedRo.

□

To prove
Property BaseConvProp1.5 that BaseConv preserves DaMod0.

Assume that P : DaMod0 and that B : Bijection obeys the preconditions for the conversion of P.

We will prove the forward implication. The reverse implication follows immediately on using B^{-1}.

As in the proof of a Move property we use the construction sequences defined in Section 4.5.4. Recall again that a construction sequence for *P* is a list of members of DaMod0, starting with EmpDaMod0 and ending with *P*. Successive elements of a construction sequence are related either by AddedEn or by AddedRo.

Form any construction sequence for *P*. For each element *P''* of the sequence we have *P''Objs* $\subseteq$ *PObjs*, *P''ConnDef* $\subseteq$ *PConnDef*, and *P''ConnRan* $\subseteq$ *PObjs* so *B* obeys the preconditions for the conversion of each element of the sequence.

Now use BaseConv with *B* to convert each element of the sequence. From BaseConvProp1.2 to 1.4 we know that the first element, EmpDaMod0, converts to EmpDaMod0 and that successive elements are still related either by AddedEn or by AddedRo. (The additional conditions for AddedRo are obeyed). Thus the last element, BaseConv(*P*, *B*), is a member of DaMod0.

We conclude that BaseConv preserves DaMod0.

□

Next we state and prove the third result : that base conversion of any member of DaMod0 gives rise to natural bijections between corresponding cartesian products. The proof uses a general result concerning cartesian products; this is stated and proved first.

We wish to describe the effect of replacing each index and each domain of a fact-style cartesian product. The replacement is described by a family of bijections, a bijection for each domain and a bijection for the cartesian product's index set. The replacement gives rise to a natural bijection between the original and transformed cartesian products. Here, "natural" means that corresponding tuples are isomorphic.

A (lengthy) property of the operator CartProd : Natural bijection when converting indexes and domains

CartProdProp2.1 .:

For any domain function F such that CartProd(F) $\neq \varnothing$,

Let $K =_{\mathrm{d}} \mathrm{Def}(F)$, (Index set)

For any family $B =_{\mathrm{d}} (B_k : \mathrm{Bijection} \mid k : (K \cup \{K\})\ )$ of bijections such that

$K \subseteq_{\mathrm{d}} B_K Dom\ \wedge$ (To define index transform)

$\forall k : K \bullet\ F_k \subseteq_{\mathrm{d}} B_k Dom$, (To define domain transforms)

Let $F' =_{\mathrm{d}} \{\ \langle B_K(k), B_k[[\ F_k\ ]]\rangle \mid k : K\ \}$, (Replace indexes and domains)

then

CartProdProp2.1a .: F' is a domain function

CartProd(F') $\neq \varnothing$

Let $C =_{\mathrm{d}}$ CartProd(F) and $C' =_{\mathrm{d}}$ CartProd(F'),

Let $B' : C \leftrightarrow C'$ be such that

$\forall T : C, T' : C' \bullet B'(T, T') \Leftrightarrow_{\mathrm{d}} \forall k : K \bullet T'Val(B_K(k)) = B_k(TVal(k))$

then

CartProdProp2.1b .: B' is uniquely determined

and

CartProdProp2.1c .: IsBijection(B')

To prove

Property CartProdProp2.1 that replacing the indexes and domain members of a cartesian product gives rise to a natural bijection.

Assume that F is a domain function such that CartProd(F) $\neq \varnothing$, and that B, F', and B' are as described in the statement of the property.

We wish to prove that F' is a domain function, that B' is uniquely determined, and that B' is a bijection.

***F′* is a domain function**

Recall from the definition of CartProd that CartProd(F) $\neq \varnothing$ requires that F is a set of couples forming the graph of a function, with $\varnothing \notin$ Ran(F). We wish to prove the same of F'.

F' is a set by a replacement axiom, of couples by definition; it is functional as B_K is an injection; and $\varnothing \notin$ Ran(F') as for each $k : K$, B_k is total, $F_k \subseteq B_k Dom$, and $F_k \neq \varnothing$. Thus CartProd(F') is not precluded from being non-empty. We now wish to prove that it is actually non-empty.

We are given that CartProd(F) is not empty. (We are not invoking the Axiom of Choice here). Take any tuple T : CartProd(F). Recall that any tuple T' : Tuple has three primary features : an index set $T'I$, a domain function

T'Domf, and a value function *T'Val*. Now form the tuple *T'* : Tuple such that $T'I =_d$ Def(*F'*) = B_K[[*K*]], *T'Domf* $=_d$ *F'*, and for each *k* : *K*, *T'Val*(B_K(*k*)) $=_d$ B_k(*TVal*(*k*)).

T' is well defined as a member of Tuple; in particular we have Def(*T'Domf*) = *T'I* = Def(*T'Val*) and for each *k'* : *T'I*, *T'Val*(*k'*) ∈ *T'Domf*(*k'*). *T'* also meets the criterion for being a member of CartProd(*F'*), namely that *T'I* = Def(*F'*) and *T'Domf* = *F'*.

We conclude that CartProd(*F'*) is not empty.

B' is uniquely determined

B' is defined by a subset axiom, so *B'* exists and is a set, and by the extensionality axiom it is uniquely determined.

B' is a bijection

We wish to prove that the relation *B'* is total, functional, surjective, and injective.

First, assume that *T* : *C* = CartProd(*F*), and that *T'* : Tuple is such that $T'I =_d$ Def(*F'*) = B_K[[*K*]], *T'Domf* $=_d$ *F'*, and for each *k* : *K*, *T'Val*(B_K(*k*)) $=_d$ B_k(*TVal*(*k*)). We have already shown that *T'* is well defined as a member of Tuple and that *T'* ∈ *C'* = CartProd(*F'*). From the definition of *B'* we also have *B'*(*T*, *T'*). We conclude that *B'* is total.

Next, assume that *T'* : *C'*, and that *T* : Tuple is such that *TI* $=_d$ Def(*F*) = *K*, *TDomf* $=_d$ *F*, and for each *k* : *K*, *TVal*(*k*) $=_d$ B_k^{-1}(*T'Val*(B_K(*k*)). By the same argument as before *T* is well defined as a member of Tuple and *T* ∈ *C*. We also have for each *k* : *K* that B_k(*TVal*(*k*)) = B_k(B_k^{-1}(*T'Val*(B_K(*k*))) so *B'*(*T*, *T'*). We conclude that *B'* is surjective.

Finally, assume that *T* : *C*, *T'* : *C'*, and that *B'*(*T*, *T'*). From the definition of *B'* and the properties of bijections it is clear that when *T* is fixed then so is *T'*, and vice versa. We conclude that *B'* is functional and injective.

Altogether, we conclude that *B'* is a bijection.

□

Now we can state and prove the BaseConv property. Recall for any *D* : DaMod0 that for each object *t* : *DObjs* the cartesian product identified by *t* is *DCart*(*t*) with domain function *DDomf*(*t*). *DCart* and *DDomf* are defined at the end of Section 4.3.2.

BaseConv Properties 2

A property of the function BaseConv when acting on members of DaMod0 : Natural bijections between corresponding Fact Types

BaseConvProp2.1 .:

$\forall D, D'$: DaMod0 $\bullet$ $\forall B$: Bijection $\bullet$
[$DRoles \subseteq BDom \ \wedge\ B[[\ DRoles\ ]] \subseteq$ Roles $\wedge$
$DEntities \subseteq BDom \ \wedge\ B[[\ DEntities\ ]] \subseteq$ Entities $\wedge$
D' = BaseConv(D, B)]
$\Rightarrow$
$\exists 1\ F : DObjs \rightarrow$ Bijection $\bullet$ (Family of bijections)
$\forall t : DObjs \bullet$
$F_t Dom = DCart(t) \ \wedge\ F_t Cod = D'Cart(B[[\ t\ ]]) \ \wedge$
$\forall T : DCart(t) \bullet \forall r : TI \bullet$
$F_t(T)Val(B(r)) = B(TVal(r))$ if $DConn(r) \subseteq$ Entities
$= F_{DConn(r)}(TVal(r))$ if $DConn(r) \subseteq$ Roles

To prove

Property BaseConvProp2.1 that base conversion of a member of DaMod0 gives a natural bijection between corresponding cartesian products.

Assume that D, D' : DaMod0 and that B : Bijection obeys the condition stated in the property. As $D \in$ DaMod0 we have $\bigcup DObjs = (DRoles \cup DEntities)$, $DConnDef = DRoles$, and $DConnRan \subseteq DObjs$ so the condition is equivalent to the usual precondition for BaseConv.

By BaseConvProp1.5 we have BaseConv(D, B) $\in$ DaMod0. Assume that D' = BaseConv(D, B).

The proof is by Well Founded induction on $DObjs$. We wish to prove that the requirements for F in the property are those of a function defined by Well Founded recursion. For this to be so we must prove that the value of F at each $t : DObjs$ is uniquely determined by t and the value of F at t's immediate predecessors.

Assume $t : DObjs$.

There are two cases to consider.

C1 $t \subseteq$ Entities

Then by definition $DCart(t) = \Phi = D'Cart(B[[\ t\ ]])$, where Φ is the nullary cartesian product $\{\phi\}$. There is exactly one bijection from $\{\phi\}$ to $\{\phi\}$. As the index set of the nullary tuple ϕ is empty the condition

$\forall T : DCart(t) \bullet \forall r : TI \bullet \ldots$

is true vacuously.

We conclude that in this case a bijection meeting the requirements for F_t exists and is uniquely determined.

C2 $t \subseteq \mathrm{Roles}$

Then t has immediate predecessors. Assume for each $x : \mathit{DPreds}(t)$ that there is a uniquely determined bijection meeting the requirements for F_x.

By definition $\mathit{DCart}(t) = \mathrm{CartProd}(\mathit{DDomf}(t))$ and $\mathit{D'Cart}(B[[\ t\ ]]) = \mathrm{CartProd}(\mathit{D'Domf}(B[[\ t\ ]]))$. We wish to prove that property CartProdProp2.1 applies here. Thus we wish to find a family of bijections that transforms the index set and domains of $\mathit{DDomf}(t)$ into those of $\mathit{D'Domf}(B[[\ t\ ]])$ in the desired way.

The index set $\mathrm{Def}(\mathit{DDomf}(t)) = t \subseteq \mathit{BDom}$, and the index set $\mathrm{Def}(\mathit{D'Domf}(B[[\ t\ ]])) = B[[\ t\ ]]$. The desired bijection for the index set is B.

For the domains of $\mathit{DDomf}(t)$ there are two possibilities for each index $r : t$. If $\mathit{DConn}(r) \subseteq \mathrm{Entities}$ then the domain is $\mathit{DConn}(r)$; the corresponding domain of $\mathit{D'Domf}(B[[\ t\ ]])$ is $\mathit{D'Conn}(B(r)) = B[[\ \mathit{DConn}(r)\ ]]$. The desired bijection for the domain in this case is B.

If $\mathit{DConn}(r) \subseteq \mathrm{Roles}$ then the domain is $\mathit{DCart}(\mathit{DConn}(r))$; the corresponding domain of $\mathit{D'Domf}(B[[\ t\ ]])$ is $\mathit{D'Cart}(\mathit{D'Conn}(B(r))) = F_{\mathit{DConn}(\mathrm{r})}[[\ \mathit{DCart}(\mathit{DConn}(r))\ ]]$ where $F_{\mathit{DConn}(\mathrm{r})}$ is the assumed bijection for this immediate predecessor of t. The desired bijection for the domain in this case is $F_{\mathit{DConn}(\mathrm{r})}$.

Thus we have the necessary bijections. We can assume that $\mathit{DCart}(t)$ is not empty, (its arity is finite), so by CartProdProp2.1 there exists a uniquely determined natural bijection from $\mathit{DCart}(t)$ to $\mathit{D'Cart}(B[[\ t\ ]])$. Furthermore, if this bijection transforms the tuple $T : \mathit{DCart}(t)$ to $T' : \mathit{D'Cart}(B[[\ t\ ]])$ then for each index $r : t$ we have $\mathit{T'Val}(B(r)) = B(\mathit{TVal}(r))$ if $\mathit{DConn}(r) \subseteq \mathrm{Entities}$ or $\mathit{T'Val}(B(r)) = F_{\mathit{DConn}(\mathrm{r})}(\mathit{TVal}(r))$ if $\mathit{DConn}(r) \subseteq \mathrm{Roles}$, which is exactly what is required, of course.

We conclude that in this case given suitable bijections for each of t's immediate predecessors then a bijection meeting the requirements for F_t exists and is uniquely determined.

Finally, from cases C1 and C2 we conclude that property BaseConvProp2.1 is true.

□

Finally, we define the relation BaseEq and sketch out a proof that it is an equivalence relation.

<u>BaseEq</u>

Predicate that is true of two members of PreMod iff they differ only in the particular roles and entities that they use

Given any $P1, P2$: PreMod then

$$
\begin{aligned}
&\text{BaseEq}(P1, P2) \Leftrightarrow_{\text{d}}\\
&\quad \exists B : \text{Bijection} \bullet\\
&\qquad [\ \forall X : (P1Objs \cup \{P1ConnDef\} \cup P1ConnRan) \bullet\\
&\qquad\quad X \subseteq BDom \wedge\\
&\qquad\quad (X \subseteq \text{Roles}) \Rightarrow B[[\ X\ ]] \subseteq \text{Roles}) \wedge\\
&\qquad\quad (X \subseteq \text{Entities}) \Rightarrow B[[\ X\ ]] \subseteq \text{Entities})\ ] \wedge\\
&\qquad P2 = \text{BaseConv}(P1, B)
\end{aligned}
$$

<u>BaseEq   Properties 1</u>

A property of the relation BaseEq

BaseEqProp1.1 .: BaseEq is an equivalence relation

Note that BaseEq requires more than a graph isomorphism on the structure of data models; it also requires the cardinality and the amount of overlap of Entity Types to be preserved.

To prove
Property BaseEqProp1.1 that BaseEq is an equivalence relation.

We wish to prove that BaseEq is reflexive, symmetric, and transitive.

Reflexive
Use the identity bijection $\text{Id}_{\text{Entities} \cup \text{Roles}}$.

Symmetric
Property BaseConvProp1.1.

Transitive
For any set X and bijections $B1$, $B2$ with $X \subseteq B1Dom$ and $B1[[\ X\ ]] \subseteq B2Dom$ we can use composition and domain, codomain restriction to give the bijection $B =_{\text{d}} X \langle\ B2 \circ B1[[\ X\ ]] \langle.\ B2 \circ B1$. Then $X \subseteq BDom$.

□

5.5 Less simple equivalences

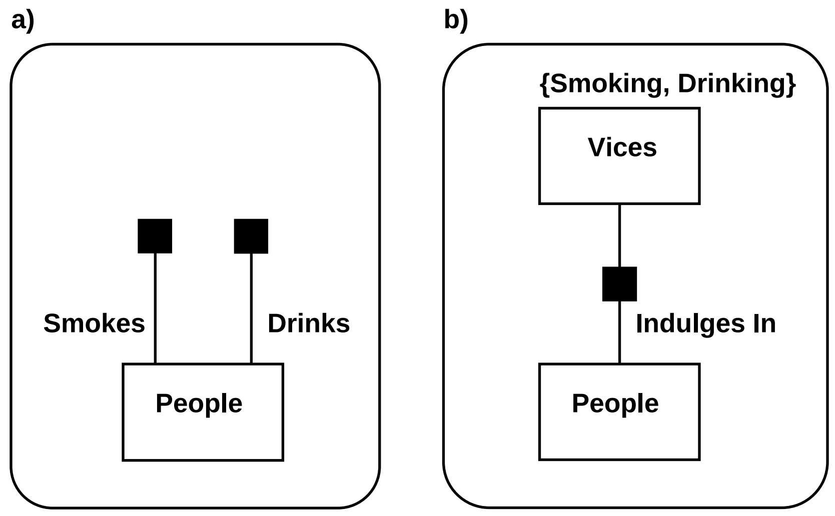
Suppose we wish to record that certain people smoke or drink. For instance we might want to record that Tom smokes, Dick drinks, and Harry both smokes and drinks. We could describe the kinds of information item we wish to record by the two generic items ' *x* smokes ' and ' *y* drinks '. Our conceptual data model would then have two unary Fact Types. Alternatively, we could describe the information items by the one generic item ' *z* indulges in *v* ' where *v* is either "smoking" or "drinking". Our data model would then have one binary Fact Type. Do the two data models and the databases they describe do the same job?

This section gives a third meaning to the word "same". Many cases have been described in detail in the literature. We will give two examples and then concentrate on a particular case that appears not to have been covered fully.

5.5.1 General

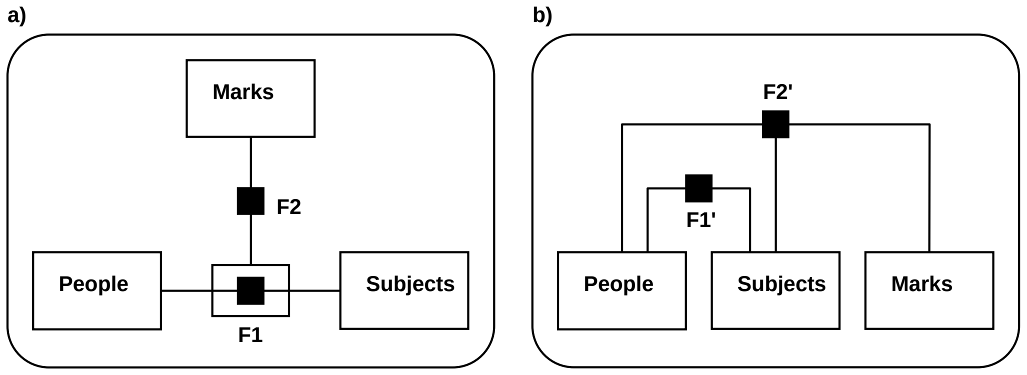
Many cases of equivalent constructions are given in Halpin [1995], p322-373. We give only two examples here. The first example is the smoking, drinking database described above. The two data models are shown in Figure 5.5.1.1(a) and (b). Clearly they do the same job; for any tuple defined in (a) there is exactly one corresponding tuple defined in (b) and vice versa. For example, for the tuple encoding ' Tom smokes ' in (a) there is the corresponding tuple encoding ' Tom indulges in smoking ' in (b).

Figure 5.5.1.1 Two ways to do it : First example



The second example uses the exam results database that we have seen several times before. Suppose we wish to record the subjects that certain students study and their exam results in those subjects. Two ways to describe a database meeting these requirements are shown in Figure 5.5.1.2(a) and (b) below.

Figure 5.5.1.2 Two ways to do it : Second example



The Fact Types, alias cartesian products, F1 and F2 defined in (a) are replaced by F1' and F2' respectively in (b).

It is obvious that F1 and F1' do the same job. For any tuple, such as the one encoding ' Carol studies Physics ', of F1 there is a corresponding tuple of F1' that does the same. If we wish to be more formal we can observe that the type structures (Section 5.3) of F1 and F1' are equivalent in the base conversion sense (Section 5.4), and that therefore there is a natural bijection between the two sets of tuples.

For each tuple of F2 there is also a corresponding tuple of F2' and vice versa. For instance, if Carol studies Physics and got 73 in the exam then there is a tuple of F2 encoding the information item ' ' Carol studies Physics ' got 73 in the exam ' and the corresponding tuple of F2' encoding ' Carol got 73 in the Physics exam '. There is a similar correspondence for any combination of person, subject, and mark, though we do not as yet have a way of saying this more formally.

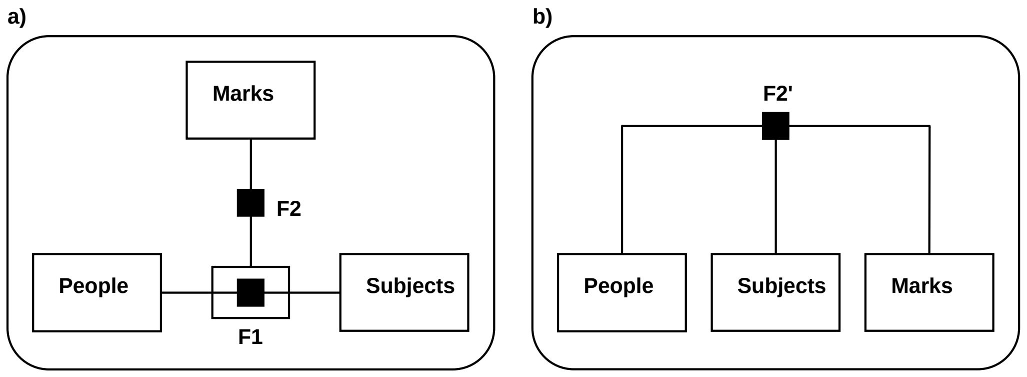
Notice that there is the implied constraint in Figure 5.5.1.2(a) that marks are restricted to recorded enrolments. This constraint would need to be made explicit in (b). (In Halpin [1995], p172-183, it is called a Subset constraint).

In going from (a) to (b) we have transformed a Fact Type, F2, from one whose domains include other Fact Types into an equivalent Fact Type, F2', whose domains are all Entity Types. This kind of transformation is called **flattening** in the literature. Flattening converts Fact Types of any rank into equivalent Fact Types of rank 1. Halpin [1995] gives a detailed treatment of flattening for Fact Types of rank 2 that have one domain that is a Fact Type, as above, but mentions the general case only in passing. A detailed treatment of the general case is desirable for reasons to be given in the next section.

The common theme in these two examples is that there is a one-to-one correspondence between the tuples defined by two data models. The correspondence need not be so simple. There could be a one-to-one correspondence between sets of tuples, as we will see in the next example, and the correspondence could be restricted to legitimate instances of the databases.

For instance, the exam database could have a business rule stating that the subject studied by a student must not be recorded until the mark is recorded. Then F1' in Figure 5.5.1.2(b) is redundant and can be deleted as in Figure 5.5.1.3 below

Figure 5.5.1.3 Two candidate designs



For each tuple of F2' defined by (b) there is now a distinct pair of tuples defined by (a), one from F1 and one from F2. We can say that (a) and (b) do the "same" job here. To complicate matters, it may be that the rule is not a business requirement but is only an option to be considered. The data model (b) is equivalent to (a) if the business rule is a definite requirement or a permitted option, but not if the rule is unacceptable.

We must conclude that whether two data models do the "same" job depends on the business context in which the databases are expected to operate. In general, testing for equivalence requires more than just the core part of the candidate data models. Consequently, we give no formal definition of this kind of equivalence for the general case.

5.5.2 Flattening : Outline

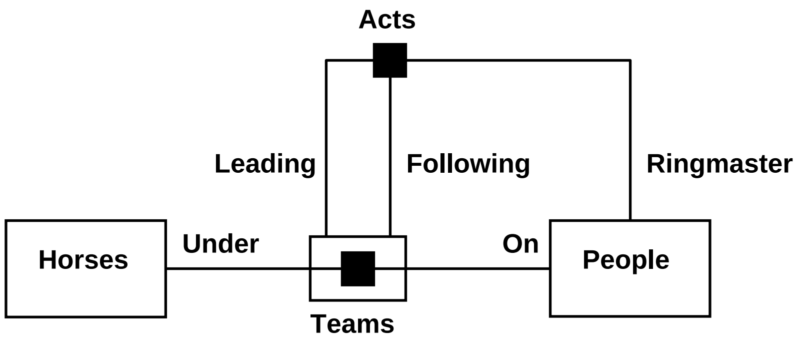
We will generalise the second example given in the previous section. We will provide a detailed description of flattening that can be applied to any Fact Type of any well-formed data model. There are two reasons for doing so.

First, conventional Relational Database Management Systems store tuples whose elements must belong to elementary data types such as numbers, character strings, and dates. The elements cannot be the arbitrary tuples that a data model can define. However, this need not restrict our use of complex Fact Types in data models. Provided Fact Types can be flattened we can use a conventional database to hold encoded versions of complex tuples. Of course, we must confirm that the original information can be recovered from the encoded version.

Second, if the flattening process transforms two dissimilar Fact Types into the same flattened Fact Type then we may have found alternative ways of meeting the same business requirements. It would be useful to have an algorithm that can test whether two Fact Types are equivalent in this sense.

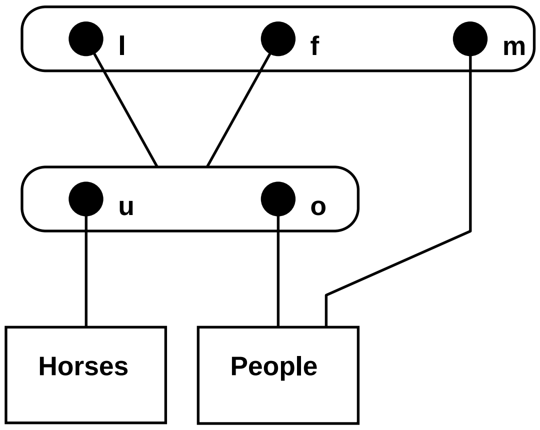
We will start with a particular example of flattening, then use it to illustrate the general case. Suppose we wish to record information about a certain kind of circus act. Each act consists of a leading team, a following team, and a ringmaster. Each team consists of a horse and a rider. For instance, one circus act has Sam as the ringmaster with Carol riding Prancer as the leading team and George riding Dancer as the following team. The beginnings of a core data model for the database is shown in Figure 5.5.2.1.

Figure 5.5.2.1 Information about a particular kind of circus act



A member of DaMod0 modelling this data model is described in Figure 5.5.2.2.

Figure 5.5.2.2 Circus acts described by a member of DaMod0

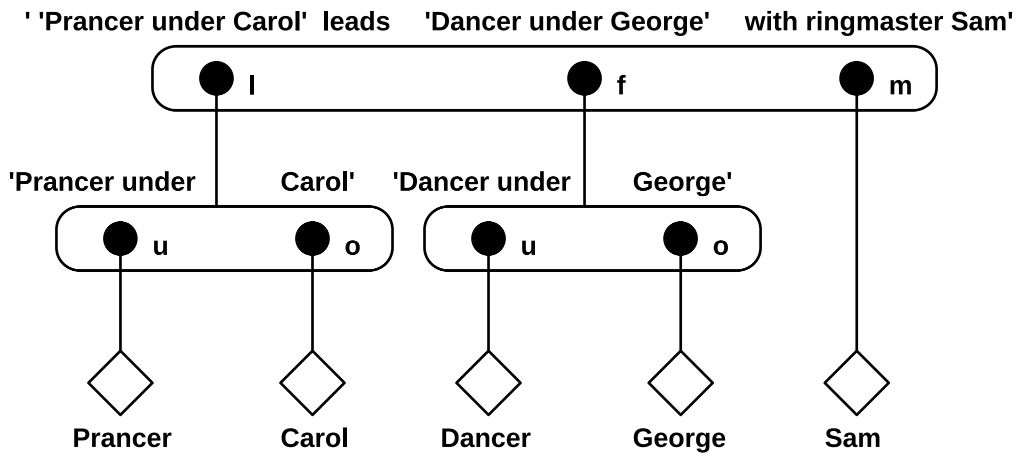


The object {l, f, m} shown in Figure 5.5.2.2 is a set of roles that identifies a cartesian product. Recall that each tuple of this cartesian product has an index set, {l, f, m}, a domain function assigning a domain to each index, and a value function assigning a value to each index. One of the tuples encodes the information item

' ' Prancer under Carol ' leads ' Dancer under George ' with ringmaster Sam '.

A picture of this tuple, or more accurately its value function, is shown in Figure 5.5.2.3 below. (Yet another ad hoc notation is used here). Notice that the indexes l and f are assigned values that are tuples belonging to the cartesian product identified by the object {u, o}.

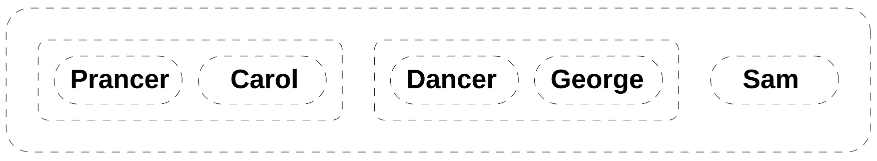
Figure 5.5.2.3 The tuple recording a particular circus act



The diamonds represent the individual entities : Prancer, Carol, Dancer, etc. Clearly, they are the only part of the tuple that differs from any other tuple of the cartesian product. The tuple has a variable part and a fixed part, just as the information item has a variable part, held in the tuple, and a fixed part. The entities could be detached and stored in any way that is convenient provided only that on retrieval they are re-attached to the correct part of the tuple.

One way to do this is to take the entities in left-to-right order in Figure 5.5.2.3 and hold them as a sequence, as in Figure 5.5.2.4. The dotted lines are there to remind us of the tuple structure; they do not represent anything that needs to be recorded repeatedly for each tuple.

Figure 5.5.2.4 The sequence recording this circus act

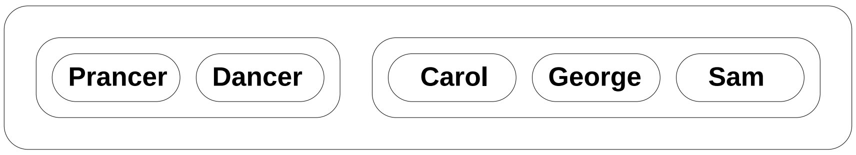


We could define this sequence to be the flattened tuple, but this would be unsatisfactory for four reasons. First, our model would then use two representations for tuples : one using the unordered index sets given in the data model, the other using ordered index sets. This would be inconvenient. Second, a database management system should insulate the application software from database reorganisations as far as possible. The DBMS should not be obliged to make orderings visible. For instance, it is allowed to supply tuples to applications as associative arrays where any underlying ordering is inaccessible. Our model should avoid using sequences where they are not essential.

Third, the Rmap procedure for transforming a data model into a database schema (Halpin [1995], p251-291) implicitly flattens all Fact Types. We should be able to show the flattened Fact Types in a data model for teaching purposes, using ordinary Fact Types with unordered index sets.

Fourth, the use of sequences can obscure useful design choices. Figure 5.5.2.5 below presents the entities in a different order. The dotted lines indicate to us that there is a Fact Type with a quite different structure that could be used to hold essentially the same information about our circus acts. We could describe an act as a horse team and a people team. The horse team has a front horse and a back horse; the people team has a leading rider, a trailing rider, and a ringmaster. The model should facilitate the recognition of such design choices.

Figure 5.5.2.5 An alternative sequence

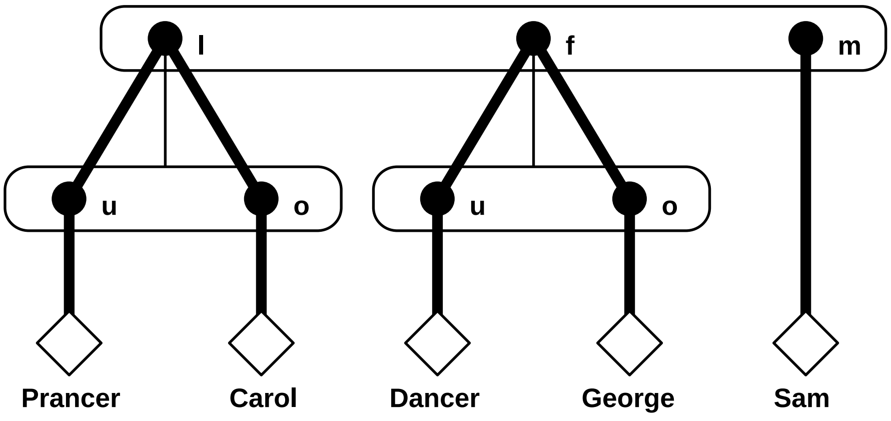


Thus when the variable part of a tuple is to be stored separately we will say that it is held as a fact-style tuple. Let us return to our example tuple, which we will call T for short. We wish to flatten T by transforming it into a fact-style tuple T' holding the entities Prancer, Carol, etc. T' has an index set, a domain function, and a value function. We are given the values : Prancer, Carol, etc. We can determine the domains from the data model : Horses for Prancer, People for Carol, etc. The non-trivial decision is to choose a convenient index set.

We wish to set two requirements for the index set. The first is that we use the same index set for each tuple of the cartesian product. Then flattening transforms one cartesian product into another. The second is that there must be an algorithm that will transform T' back into T.

The picture of T is reproduced in Figure 5.5.2.6 below. Observe that if we start at any entity, say Dancer, then we can follow a path upwards from role to role. For Dancer we visit Dancer, then u, then f. This path is unique to Dancer. If we remove Dancer leaving us with u then f we have an "almost path" that is also unique to Dancer. Recall that the index sets occurring in any member of DaMod0 are pairwise disjoint. Consequently if we record the roles occurring in the almost path then we have a set, {u, f}, that is unique to Dancer. Thus {u, f} is a set of roles that identifies the unique path starting at Dancer. The sets of roles formed in this way will be our indexes.

Figure 5.5.2.6 The paths in the forest



Each index identifies a path that visits exactly one entity. If we map each index to its entity then we have the value function of the flattened tuple T', see Figure 5.5.2.7.

Figure 5.5.2.7 The value function

$$\left\{ \begin{tikzcd} \{l, u\} \arrow[d, mapsto] & \{l, o\} \arrow[d, mapsto] & \{f, u\} \arrow[d, mapsto] & \{f, o\} \arrow[d, mapsto] & \{m\} \arrow[d, mapsto] \\ \text{Prancer} & \text{Carol} & \text{Dancer} & \text{George} & \text{Sam} \end{tikzcd} \right\}$$

It is now straightforward to construct the domain function, see Figure 5.5.2.8.

Figure 5.5.2.8 The domain function

$$\left\{ \begin{tikzcd} \{l, u\} \arrow[d, mapsto] & \{l, o\} \arrow[d, mapsto] & \{f, u\} \arrow[d, mapsto] & \{f, o\} \arrow[d, mapsto] & \{m\} \arrow[d, mapsto] \\ \text{Horses} & \text{People} & \text{Horses} & \text{People} & \text{People} \end{tikzcd} \right\}$$

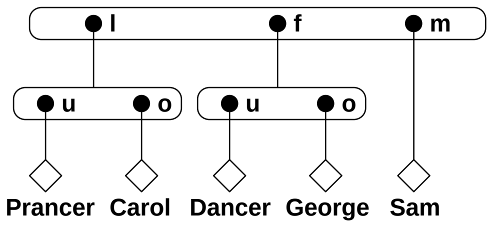
We have now defined an index set, a domain function, and a value function : we have our flattened tuple T'. Notice that if we are given T' then we can reconstruct the tuple T. Each index of T' identifies a particular path in every tuple of the original cartesian product. The value function of T' selects a particular value at the end of each path, so identifying a unique tuple, T.

Now we can describe the general case. As might be expected, flattening will be defined by recursion but we must show how this can be done first. Assume that we have any member *D* of DaMod0 and any object *t* of *DObjs*. If *t* is a set of entities then there are no almost paths ending at *t* and there is nothing to be done. We do not flatten entities.

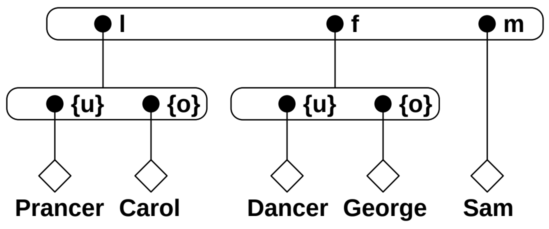
Now assume that *t* is a set of roles. *t* identifies the cartesian product *DCart*(*t*). We wish to flatten its tuples. The general process is illustrated below in Figure 5.5.2.9 for the case of our example tuple T. Recall that the index set of each tuple of *DCart*(*t*) is *t* itself. Each role belonging to *t* is at the end of one or more paths and so contributes one or more indexes to the index set of each flattened tuple and one or more maplets to its value function.

Figure 5.5.2.9 Flattening a tuple

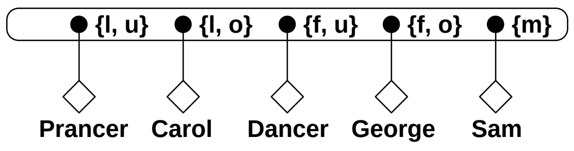
a) Unflattened tuple T



b) Tuple partly flattened



c) Flattened tuple T'



Consider any role *r* belonging to the object *t*.

If the data model associates *r* with a set of entities, that is if *DConn*(*r*) $\subseteq$ Entities, then *r* contributes only one index, {*r*}, to the index set of each flattened tuple. This is the case for the role m in Figure 5.5.2.9 where the one index is {m}. The value function of any tuple of *DCart*(*t*) contains the maplet $r \mapsto e$ for some entity *e* : *DConn*(*r*). Flattening transforms this maplet to $\{r\} \mapsto e$. In Figure 5.5.2.9 the maplet m $\mapsto$ Sam is transformed to the maplet {m} $\mapsto$ Sam. Likewise, the domain function of any tuple of *DCart*(*t*) contains the maplet $r \mapsto$ *DConn*(*r*). Flattening transforms this maplet to $\{r\} \mapsto$ *DConn*(*r*).

On the other hand, if the data model associates *r* with a set *x* of roles, that is if *x* = *DConn*(*r*) $\subseteq$ Roles, then there is one path ending at *r* for each path ending at some role of *x*. If a path ending at a role of *x* is identified by the set *p* of roles then the corresponding path ending at *r* is identified by $p \cup \{r\}$. Thus the indexes contributed by *r* can be derived from the index set we use when we flatten the tuples of the cartesian product *DCart*(*x*). This is the case for the role l in Figure 5.5.2.9 where the two indexes {u} and {o} are extended to give the indexes {l, u} and {l, o}.

The value function of any tuple of *DCart*(*t*) contains the maplet $r \mapsto$ *T1* for some tuple *T1* : *DCart*(*x*). Flattening transforms this maplet to one or maplets that can be derived from the maplets we get by flattening *T1* to give *T1'*. If the value function of *T1'* contains the maplet $p \mapsto e$ for some entity *e* then we derive the maplet $(p \cup \{r\}) \mapsto e$. In

Figure 5.5.2.9 the maplets {u} $\mapsto$ Prancer and {o} $\mapsto$ Carol are transformed to the maplets {l, u} $\mapsto$ Prancer and {l, o} $\mapsto$ Carol. Likewise, the domain function of any tuple of *DCart*(*t*) contains the maplet $r \mapsto$ *DCart*(*x*). Flattening transforms this maplet to one or more maplets that can be derived from any flattened tuple of *DCart*(*x*), such as *T1'*. If the domain function of *T1'* contains the maplet $p \mapsto E$ for some domain *E* then we derive the maplet $(p \cup \{r\}) \mapsto E$.

We have now defined a flattening function which given any tuple of *DCart*(*t*) returns a well defined flattened tuple. (This is confirmed in the details section). Notice that this function has been defined in terms of *t* and of the flattening functions for the immediate predecessors of *t*. The definition fits the pattern of definition by Well Founded recursion on the members of *DObjs* as described in Section 4.3.1. We can be sure that there is a uniquely determined flattening function $DFlatten_t$ for each object *t* : *DObjs*, and that this is so for each member *D* of DaMod0. We can conclude that each tuple of each Fact Type of each well formed data model can be flattened and hence can be represented in a conventional database (provided sufficient storage space is available, of course).

We must also define unflattening functions. The principles are much the same as for flattening so we will merely illustrate it with our circus act example. Refer back to Figure 5.5.2.9. We wish to transform the flattened tuple T' back into T. To do this we must assign a value to each role of the index set {l, f, m}. For the role m we know that T' has the index {m}. T assigns the same value, Sam, to m as T' assigns to {m}.

For the role l we know that T' has one or more indexes containing l, namely {l, u} and {l, o}. From the maplets {l, u} $\mapsto$ Prancer and {l, o} $\mapsto$ Carol we can form the flattened tuple whose value function consists of the maplets {u} $\mapsto$ Prancer and {o} $\mapsto$ Carol, as in Figure 5.5.2.9(b). Unflattening this tuple gives us the value that T assigns to l. Do the same for the role f.

As with flattening, we can be sure that there is a uniquely determined unflattening function $DUnflatten_t$ for each object *t* : *DObjs*, and that this is so for each member *D* of DaMod0.

We can go a little further and define several related features. Recall from Section 4.3 that any domain function *F* can be used to define a cartesian product. CartProd(*F*) is the set of all fact-style tuples whose domain function is *F*. The features are as follows.

For each *D* : DaMod0, *t* : *DObjs*

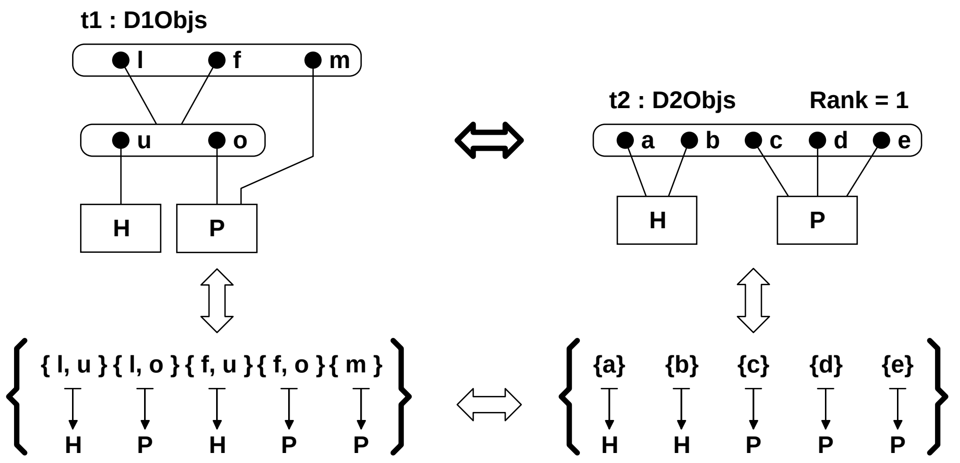
DFlatIndex(*t*)	is the index set used when flattening the tuples of *DCart*(*t*);
$DFlatDomf_t$	is the domain function used when flattening the tuples of *DCart*(*t*);
DFlatCart(*t*)	is CartProd($DFlatDomf_t$), a cartesian product;
DFlatArity(*t*)	is the arity of the tuples we get when we flatten the tuples of *DCart*(*t*);
$DFlatten_t$	is the function that flattens any tuple of *DCart*(*t*);
$DUnflatten_t$	is the function that unflattens any tuple of *DFlatCart*(*t*).

For any *D* : DaMod0 and *t* : *DObjs* we would expect, and in the details section we prove, that $DFlatten_t$ is a one-to-one correspondence (a bijection) from *DCart*(*t*) to *DFlatCart*(*t*) and that $DUnflatten_t$ is its inverse. The two cartesian products are equivalent in an obvious sense. If tuples are stored in a database in their flattened form then the original tuples can always be recovered.

We have shown that all the tuples defined by any well-formed data model can be implemented by a conventional DBMS. We will now turn to two topics of more direct interest to data modellers. First, we will show that the flattened tuples of any Fact Type can be represented in the data model as a (derived) Fact Type. Second, we will define a test to decide whether two Fact Types, nested to any degree, are able to meet the same business requirements. If they are, then deciding which is preferable will be a matter of judgement, of course.

We cannot claim that the index set of a flattened cartesian product is a set of roles, so we cannot claim that a flattened cartesian product is a Fact Type in the sense used for DaMod0. However, we can define a proper Fact Type that is equivalent. An example is given in Figure 5.5.2.10. The upper half of the picture shows D1, D2 : DaMod0 with the objects t1 : D1Objs and t2 : D2Objs marked. The lower half shows two flattened domain functions : D1FlatDomf_{t1} on the left and D2FlatDomf_{t2} on the right.

Figure 5.5.2.10 An equivalent Fact Type of rank 1



We know that there is a bijection from D1Cart(t1) to D1FlatCart(t1) and from D2FlatCart(t2) to D2Cart(t2). In spite of the domains being listed in different orders we observe that the flattened domain function on the left can be transformed into the one on the right by replacing indexes : replace {l, u} by {a}, {l, o} by {c}, {f, u} by {b}, etc. Therefore there is a natural bijection from D1FlatCart(t1) = CartProd(D1FlatDomf_{t1}) to D2FlatCart(t2) = CartProd(D2FlatDomf_{t2}) that preserves the entities occurring in each tuple. We now have three bijections which we can compose to give us a bijection from D1Cart(t1) to D2Cart(t2). The two cartesian products are equivalent. We can say that the Fact Type identified by t2 represents the flattened tuples of the Fact Type identified by t1.

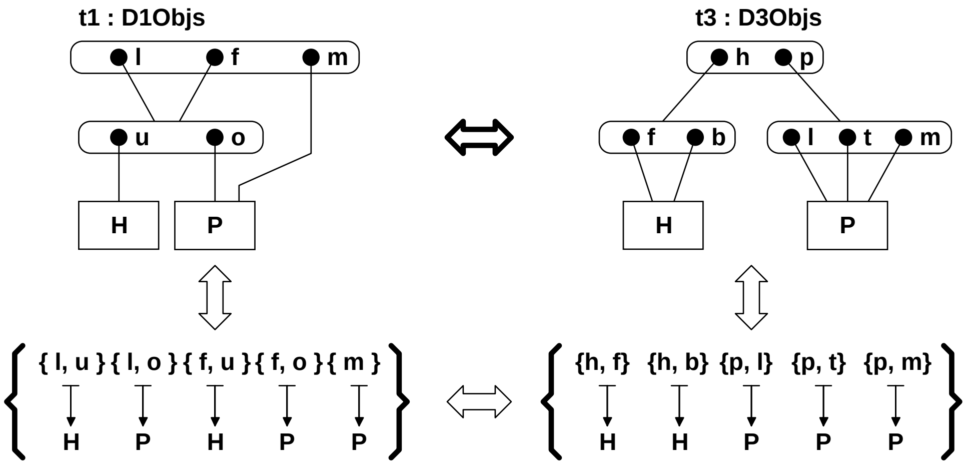
Each index of the index set D2FlatIndex(t2) is of the form {*r*} for some role *r*. Notice that this gives the flattened domain function a special property. From the flattened domain function we can reconstruct t2, its type structure, and the Fact Type it identifies without ambiguity and without reference to any member of DaMod0 or any data model.

In other words, given the object t1 occurring in D1 we are able to construct an object t2 and a D2 : DaMod0 able to represent the flattened tuples of the Fact Type identified by t1. D2 could be the result of adding t2 and its role associations to D1, if desired.

We can do this for any object of any member of DaMod0, provided that the object is a set of roles and that there are sufficient roles available. Doing it for Entity Types is not possible, of course. When a data model is transformed into the schema for a conventional database we can say that the first step is to translate each Fact Type into its flattened equivalent in this way.

We will now generalise this construction by allowing both objects to be of any rank. An example is given in Figure 5.5.2.11.

Figure 5.5.2.11 Two equivalent Fact Types


t1 : D1Objs
l
f
m
u
o
H
P
t3 : D3Objs
h
p
f
b
l
t
m
H
P
{ l, u } { l, o } { f, u } { f, o } { m }
H P H P P
{h, f} {h, b} {p, l} {p, t} {p, m}
H H P P P

We claim that there is a natural bijection from D1Cart(t1) to D3Cart(t3). We prove this by observing that there is a bijection between the two flattened index sets that preserves domains. For instance {l, u} to {h, f}, {l, o} to {p, l}, {f, u} to {h, b}, {f, o} to {p, t}, and {m} to {p, m}. Consequently there is a natural bijection from D1FlatCart(t1) to D3FlatCart(t3) that preserves entity occurrences in the tuples, and hence there is one from D1Cart(t1) to D3Cart(t3). Incidentally, although there are several domain-preserving bijections between the flattened index sets, once this bijection is fixed then so is the natural bijection between the two cartesian products.

We have confirmed what Figure 5.5.2.5 suggested to us : that there is an alternative description of circus acts that we might wish to consider. Whether t1 or t3 is the better design depends on business requirements that are not available to us here.

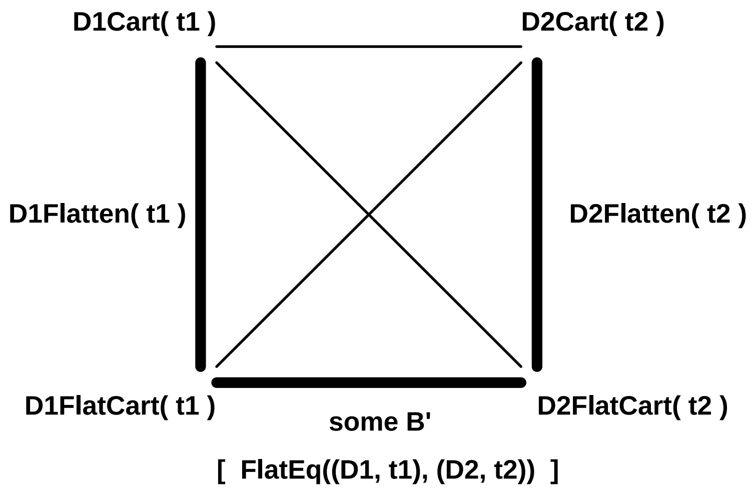
We wish to be able to test for this kind of equivalence for any pair of objects. We define the relation FlatEq to do this. For any *D1, D2* : DaMod0, *t1* : *D1Objs*, *t2* : *D2Objs* then FlatEq((*D1*, *t1*), (*D2*, *t2*)) is true iff there is a bijection from *D1FlatIndex*(*t1*) to *D2FlatIndex*(*t2*) that preserves domains in the flattened domain functions. If FlatEq((*D1*, *t1*), (*D2*, *t2*)) is true then there is a natural bijection from the Fact Type *D1Cart*(*t1*) to the Fact Type *D2Cart*(*t2*).

We have two examples of equivalence. In Figure 5.5.2.11 we have FlatEq((D1, t1), (D3, t3)) and in Figure 5.5.2.10 we have FlatEq((D1, t1), (D2, t2)). One consequence of the detailed definition of FlatEq is that any two Entity Types are equivalent in the FlatEq sense. This is somewhat artificial, though it should remind us that entities are not independent units of information in database instances.

Observe that there is a way to implement FlatEq that avoids the need to construct flattened indexes. For *t1* : *D1Objs* follow each path downwards from the roles of *t1*. Count the number of occurrences of each Entity Type. Do the same for *t2* : *D2Objs*. We have FlatEq((*D1, t1*), (*D2, t2*)) iff for each Entity Type *E* the number of occurrences of *E* is the same for *t1* as for *t2*. (We compare two bags).

The network of bijections that arises when two objects are equivalent in the FlatEq sense is shown in Figure 5.5.2.12. The highlighted lines represent the bijections that have been described explicitly, and the other lines represent derived bijections. Remember that the bottom bijection *B'* is not necessarily unique.

Figure 5.5.2.12 The network of bijections when FlatEq((D1, t1), (D2, t2))



This completes our third definition of the word "same", but we must remember that we have defined it for only one of the many cases that can arise in practice.

5.5.3 Flattening : Details

In this section we define features concerning flattening and the equivalence relation derived from them.

We start by defining some features of each member of DaMod0 that provide flattened index sets, domain functions, cartesian products, and arities. Recall from Section 4.3.2 that a domain function is a set of couples and that the operator CartProd returns a fact-style cartesian product on being given a domain function.

<u>*D* : DaMod0   Secondary features 6</u>

Some secondary features of each member of DaMod0 : Flattening

DFlatIndex	Index sets
$DFlatIndex(t) \subseteq_{d}$ Pow(*Roles*) (*t* : *DObjs*)	The identifiers of the "almost paths" starting at *t*. I.e the index set of the flattened cartesian product identified by *t*

DDefFlatIndex .: $\forall t : DObjs$ •
If $DPreds(t) = \varnothing$ (i.e $t \subseteq$ Entities)
then $DFlatIndex(t) =_{d} \varnothing$

else $DFlatIndex(t) =_{d}$ (i.e $t \subseteq$ Roles)
$\{ \{r\} \mid r : t \wedge DConn(r) \subseteq \text{Entities} \} \cup$
$\{ p + \{r\} \mid r : t \wedge DConn(r) \subseteq \text{Roles} \wedge p : DFlatIndex(DConn(r)) \}$

DFlatDomf	Domain functions
$DFlatDomf_{t}$ (*t* : *DObjs*)	The domain function (as a set of couples) of the flattened cartesian product identified by *t*

DDefFlatDomf .: $\forall t : DObjs$ •
$DFlatDomf_{t} =_{d}$
$\{ \langle p, DConn(s) \rangle \mid p : DFlatIndex(t) \wedge s : p \wedge DConn(s) \subseteq \text{Entities} \}$

DFlatCart	Cartesian products
$DFlatCart(t) \subseteq_{d}$ Tuple (*t* : *DObjs*)	Flat cartesian product identified by *t*

DDefFlatCart .: $\forall t : DObjs$ • $DFlatCart(t) =_{d} \text{CartProd}(DFlatDomf_{t})$

DFlatArity	Arities
$DFlatArity(t)$: Nat (*t* : *DObjs*)	The arity of *DFlatCart*(*t*)

DDefFlatArity .: $\forall t : DObjs$ • $DFlatArity(t) =_{d} \text{Num}(DFlatIndex(t))$

We state and prove some properties. The two main results are that flattened cartesian products have been properly defined (*DProp7.13*), and that there is an alternative recursive definition for their arities (*DProp7.16*). The proofs make use of several minor results that are stated and proved separately.

D : DaMod0 Properties 7

Some properties of each member of DaMod0

DProp7.1 .: $\forall t : DObjs \bullet DFlatIndex(t) = \varnothing \Leftrightarrow t \subseteq \text{Entities}$

DProp7.2 .: $\forall t : DObjs \bullet \forall p : DFlatIndex(t) \bullet p \subseteq DRoles \wedge p \subseteq \bigcup DAncIds(t)$

DProp7.3 .: $\forall t : DObjs \bullet$
$\text{IsFinite}(DFlatIndex(t)) \wedge \forall p : DFlatIndex(t) \bullet \text{IsFinite}(p)$

DProp7.5 .: $\forall t : DObjs \bullet \forall p : DFlatIndex(t) \bullet \exists 1\ r \bullet r \in (p \cap t)$

DProp7.6 .: $\forall t : DObjs \bullet \forall p : DFlatIndex(t) \bullet \exists 1\ s : p \bullet DConn(s) \subseteq \text{Entities}$

DProp7.9 .: $\forall t : DObjs \bullet \text{Def}(DFlatDomf_t) = DFlatIndex(t)$

DProp7.13 .: $\forall t : DObjs \bullet DFlatCart(t) \neq \varnothing$

DProp7.16 .: Alternative definition of *DFlatArity*

$$
\begin{array}{l}
\forall t : DObjs \bullet \\
\quad DFlatArity(t) = \begin{array}[t]{ll}
\text{if} & t \subseteq \text{Entities} \\
\text{then} & 0 \\
\text{else} & \Sigma_{r\,:\,t}\ (\ \begin{array}[t]{ll}
\text{if} & DConn(r) \subseteq \text{Entities} \\
\text{then} & 1 \\
\text{else} & DFlatArity(DConn(r))\ )
\end{array}
\end{array}
\end{array}
$$

Property *DProp7.16* is immediate from the definitions of *DFlatArity* and *DFlatIndex* and the finiteness property *DProp7.3*. *DProp7.9* is immediate from *DProp7.6* and the definition of *DFlatDomf*.

Properties *DProp7.1*, *7.2*, *7.3*, *7.5*, and *7.6* are proved together and *DProp7.13* is proved separately.

To prove
Properties *DProp7.1*, *7.2*, *7.3*, *7.5*, and *7.6* concerning the "almost paths" that are used as the indexes of flattened tuples.

Let $\alpha =_{\text{SYM}}$ $\text{IsFinite}(DFlatIndex(t)) \wedge [\ DFlatIndex(t) = \varnothing \Leftrightarrow t \subseteq \text{Entities}\ ] \wedge$
$\forall p : DFlatIndex(t) \bullet \beta$

Let $\beta =_{\text{SYM}}$ $p \subseteq DRoles \wedge p \subseteq \bigcup DAncIds(t) \wedge \text{IsFinite}(p) \wedge$
$[\ \exists 1\ r \bullet r \in (p \cap t)\ ] \wedge [\ \exists 1\ s : p \bullet DConn(s) \subseteq \text{Entities}\ ]$

Let $\alpha' =_{\text{d}} \alpha^t_x$, $\beta' =_{\text{d}} \beta^t_x$, meaning x is substituted for t in α and β.

We wish to prove that $\forall D : \text{DaMod0} \bullet \forall t : DObjs \bullet \alpha$.

Assume that D : DaMod0 and $t : DObjs$.

The proof is by Well Founded induction on *DObjs*. There are two cases to consider.

C1 Case $DPreds(t) = \emptyset$

Then $t \subseteq$ Entities and $DFlatIndex(t) = \emptyset$. $DFlatIndex(t)$ is finite and $\forall p : DFlatIndex(t) \bullet \beta$ is true vacuously.

We conclude that in this case α is true.

C2 Case $DPreds(t) \neq \emptyset$

Then $t \subseteq DRoles \subseteq$ Roles and t is finite. Assume for each $x : DPreds(t)$ that α' is true.

From its definition we see that each index belonging to $DFlatIndex(t)$ is generated by a role $r : t$. Consider the two cases for r.

C2.C1 Case $DConn(r) \subseteq$ Entities

Then r generates the one index $\{r\}$. For this index we have $\{r\} \subseteq DRoles$, $\{r\} \subseteq t \subseteq \bigcup DAncIds(t)$, and $\{r\}$ is finite.

Also, r is the only member of $(\{r\} \cap t)$, and r is the only member of $\{r\}$ for which $DConn(r) \subseteq$ Entities.

We conclude that if $p = \{r\}$ then β is true.

C2.C2 Case $DConn(r) \subseteq$ Roles

Then r generates the index $(p' + \{r\})$ for each $p' : DFlatIndex(DConn(r))$. By α', we have $DFlatIndex(DConn(r)) \neq \emptyset$. We also have

$$p' \subseteq \bigcup DAncIds(DConn(r)) \subseteq \bigcup DAncs(t).$$

t is disjoint from each of its ancestors so $r \notin p'$, justifying the use of "+" to indicate a union of disjoint sets.

From the properties of p' implied by α' we can deduce the desired properties of the index $(p' + \{r\})$:

$p' \subseteq DRoles$ and $\{r\} \subseteq t \subseteq DRoles$ so $(p' + \{r\}) \subseteq DRoles$;

$(p' + \{r\}) \subseteq (\bigcup DAncs(t)) \cup t = \bigcup DAncIds(t)$;

p' is finite so $(p' + \{r\})$ is finite;

$p' \subseteq \bigcup DAncs(t))$ so $p' \cap t = \emptyset$,

hence r is the only member of $(p' + \{r\}) \cap t$;

As $DConn(r) \subseteq$ Roles

there is only one member $s : (p' + \{r\})$ for which $DConn(s) \subseteq$ Entities.

We conclude that if $p = (p' + \{r\})$ then β is true.

We conclude from cases C2.C1 and C2.C2 that $\forall p : DFlatIndex(t) \bullet \beta$.

Continuing case C2, for each $x : DPreds(t)$ we have by α' that $DFlatIndex(x)$ is finite and that $DFlatIndex(x) = \emptyset$ iff $x \subseteq$ Entities. Now t is finite and each member of t generates a finite number of indexes, so $DFlatIndex(t)$ is finite. Also t is not empty so the members of t generate one index each in case C2.C1 and at least one index each in case C2.C2. Hence $DFlatIndex(t) \neq \emptyset$.

We conclude in case C2 that $(\forall x : DPreds(t) \bullet \alpha') \Rightarrow \alpha$.

Finally, from case C1 and C2 we conclude that $\forall D :$ DaMod0 $\bullet$ $\forall t : DObjs \bullet \alpha$.

To prove

Property *DProp7.13* that *DFlatCart* is properly defined. That is, to prove that

$\forall D$: DaMod0 $\bullet$ $\forall t : DObjs \bullet DFlatCart(t) \neq \emptyset$.

Assume that D : DaMod0 and $t : DObjs$.

We wish to prove that $DFlatDomf_t$ obeys the preconditions for the operator CartProd and that the index set of $DFlatDomf_t$ is finite. Then we can be sure that CartProd($DFlatDomf_t$) = $DFlatCart(t)$ is not empty.

Thus we wish to prove that $DFlatDomf_t$ is a set of couples forming the graph of a function from indexes to non-empty domains, and that the index set is finite. Recall the definition :

$DFlatDomf_t =_{\mathrm{d}}$

$\{ \langle p, DConn(s) \rangle \mid p : DFlatIndex(t) \ \wedge \ s : p \ \wedge \ DConn(s) \subseteq$ Entities $\}$.

Note that if $t \subseteq$ Entities then by *DProp7.1* $DFlatIndex(t) = \emptyset$ and the following statements are vacuously true.

Set of couples

$DFlatDomf_t$ is a set by a replacement axiom, of couples by construction.

Functional

By *DProp7.6* there is at most one member, s, of each $p : DFlatIndex(t)$ for which $DConn(s) \subseteq$ Entities.

Non-empty domains

Each index is associated with a member of *DConnRan*, and hence with a member of *DObjs*. Every member of *DObjs* is non-empty.

Finite index set

The index set is a subset of $DFlatIndex(t)$ which by *DProp7.3* is finite.

□

Next we define some features of each member of DaMod0 that flatten and unflatten tuples. Recall from Section 4.3.2 that any fact-style tuple T has three features : an index set TI, a domain function $TDomf$, and a value function $TVal$; for each index $i : TI$, $TVal(i)$ is constrained to be a member of $TDomf(i)$.

<u>D : DaMod0   Secondary features 7</u>

Some secondary features of each member of DaMod0 : More flattening

DFlatten	Flattening functions
$DFlatten_t : DCart(t) \rightarrow DFlatCart(t)$ ($t : DObjs$)	Natural bijection

$DDefFlatten$.: $\forall t : DObjs \bullet \forall T : DCart(t) \bullet$

$DFlatten_t(T) = T'$ where $T' : DFlatCart(t)$ and

$\forall p : DFlatIndex(t) \bullet \forall r : (p \cap t) \bullet$ (Note : $\exists 1 r : (p \cap t)$)

$[\ p = \{r\} \ \Rightarrow \ T'Val(p) = TVal(r)\ ] \ \wedge$

$[\ p \neq \{r\} \ \Rightarrow \ T'Val(p) = DFlatten_{DConn(r)}(TVal(r))Val(\ p - \{r\})\ ]$

DFlatPart Auxiliary functions for unflattening

$DFlatPart_t(r, T) : DFlatCart(DConn(r))$

$(t : DObjs,\ r : t \cap \text{Roles},\ T : DFlatCart(t))$

$DDefFlatPart$.: $\forall t : DObjs \bullet \forall r : t \cap \text{Roles} \bullet \forall T : DFlatCart(t) \bullet$

$DFlatPart_t(r, T) = T'$ where $T' : DFlatCart(DConn(r))$ and

$\forall p : DFlatIndex(DConn(r)) \bullet T'Val(p) = TVal(p + \{r\})$

DUnflatten Unflattening functions

$DUnflatten_t : DFlatCart(t) \to DCart(t)$ Natural bijection

$(t : DObjs)$

$DDefUnflatten$.: $\forall t : DObjs \bullet \forall T' : DFlatCart(t) \bullet$

$DUnflatten_t(T') = T$ where $T : DCart(t)$ and

$\forall r : (t \cap \text{Roles}) \bullet$

$[\, DConn(r) \subseteq \text{Entities} \Rightarrow TVal(r) = T'Val(\{r\})\,] \wedge$

$[\, DConn(r) \subseteq \text{Roles} \Rightarrow TVal(r) = DUnflatten_{DConn(r)}(DFlatPart_t(r, T'))\,]$

We wish to prove that flattening and unflattening are described by well defined bijections that are inverses of each other. This is done in three stages.

D : DaMod0 Properties 8

Some properties of each member of DaMod0

$DProp8.1$.: $\forall t : DObjs \bullet$

$DFlatten_t$ is well defined as a member of $DCart(t) \to DFlatCart(t)$

$DProp8.2$.: $\forall t : DObjs \bullet \forall r : t \cap \text{Roles} \bullet$

$DFlatPart_{t,\, r}$ is well defined as a member of

$DFlatCart(t) \to DFlatCart(DConn(r))$

$DProp8.3$.: $\forall t : DObjs \bullet$

$DUnflatten_t$ is well defined as a member of $DFlatCart(t) \to DCart(t)$

$DProp8.4$.: $\forall t : DObjs \bullet \forall T : DCart(t) \bullet \forall r : t \cap \text{Roles} \bullet$

$DConn(r) \subseteq \text{Roles} \Rightarrow DFlatPart_t(r, DFlatten_t(T)) = DFlatten_{DConn(r)}(TVal(r))$

$DProp8.5$.: $\forall t : DObjs \bullet \forall T : DCart(t) \bullet DUnflatten_t(DFlatten_t(T)) = T$

$DProp8.6$.: $\forall t : DObjs \bullet \forall T' : DFlatCart(t) \bullet DFlatten_t(DUnflatten_t(T')) = T'$

$DProp8.7$.: $\forall t : DObjs \bullet$

$\text{IsBijection}(DFlatten_t) \wedge \text{IsBijection}(DUnflatten_t) \wedge DUnflatten_t = DFlatten_t^{-1}$

To prove

Property *DProp8.1* that flattening is well defined. That is, to prove that

$\forall D : \text{DaMod0} \bullet \forall t : DObjs \bullet$

The definition of $DFlatten_t$ specifies exactly one member of

$DCart(t) \to DFlatCart(t)$.

Assume that D : DaMod0 and $t : DObjs$.

Recall the definition : $\forall T : DCart(t) \bullet$

$DFlatten_t(T) = T'$ where $T' : DFlatCart(t)$ and

$\forall p : DFlatIndex(t) \bullet \forall r : (p \cap t) \bullet$

$[\, p = \{r\} \Rightarrow T'Val(p) = TVal(r)\,] \wedge$

$[\, p \neq \{r\} \Rightarrow T'Val(p) = DFlatten_{DConn(r)}(TVal(r))Val(\, p - \{r\})\,]$

The proof is by Well Founded induction on $DObjs$. There are two cases to consider.

C1 Case $t \subseteq$ Entities

Then by *DProp7.1* $DFlatIndex(t) = \varnothing$ so $DFlatCart(t) = \Phi = DCart(t)$ where Φ is the nullary cartesian product $\{\phi\}$. The only total function from Φ to Φ, which maps ϕ to ϕ, obeys the definition, vacuously in part.

We conclude that $DFlatten_t$ is well defined.

C2 Case $t \subseteq$ Roles

Then by *DProp7.1* $DFlatIndex(t) \neq \varnothing$ and by *DProp7.13* $DFlatCart(t) \neq \varnothing$.

We wish to prove that for each tuple of $DCart(t)$ there is exactly one tuple of $DFlatCart(t)$ that obeys the definition. For this to be so the definition must assign exactly one value to each index belonging to $DFlatIndex(t)$, and the value must belong to the domain indicated by $DFlatDomf_t$.

Assume that $T : DCart(t)$ and $p : DFlatIndex(t)$. Recall that every fact-style tuple has a value function, and that the function is defined at each of the tuple's indexes. Recall also that the index set of T is t.

By *DProp7.5* $p \cap t$ is a singleton set, say $\{r\}$. Either $p = \{r\}$ or it does not. Consider these two cases.

C2.C1 Case $p = \{r\}$

Then by *DProp7.6* $DConn(r) \subseteq$ Entities and from their definitions $DFlatDomf_t(p) = DConn(r) = DDomf_t(r)$. p is assigned the value $TVal(r) \in DDomf_t(r) = DFlatDomf_t(p)$.

We conclude that p is assigned exactly one value and it belongs to the correct domain.

C2.C2 Case $p \neq \{r\}$

Then from the definition of $DFlatIndex$ $DConn(r) \subseteq$ Roles, (for if not we would have $p = \{r\}$), so $p - \{r\} \in DFlatIndex(DConn(r))$.

Assume for each $x : DPreds(t)$ that $DFlatten_x$ is a uniquely determined total function from $DCart(x)$ to $DFlatCart(x)$. Then in particular this is so when $x = DConn(r)$.

By *DProp7.6* there is exactly one member, say s, of p such that $DConn(s) \subseteq$ Entities. By its definition $DFlatDomf_t(p) = DConn(s)$. As s cannot equal r then $s \in p - \{r\}$.

By the definition of $DCart$ and $DDomf$ the value $TVal(r) \in DDomf_t(r) = DCart(DConn(r))$. Then by assumption $DFlatten_{DConn(r)}$ assigns exactly one tuple of $DFlatCart(DConn(r))$ to $TVal(r)$. Call this tuple T''. p is assigned the value $T''Val(\ p - \{r\})$. By the definition of *DFlatDomf* the value $T''Val(\ p - \{r\}) \in DFlatDomf_{DConn(r)}(\ p - \{r\}) = DConn(s) = DFlatDomf_t(p)$.

We conclude that p is assigned exactly one value and it belongs to the correct domain.

Continuing case C2, from case C2.C1 and C2.C2 we conclude that each p is assigned exactly one value of the correct domain, and hence we conclude that $DFlatten_t$ is well defined, provided that $\forall x : DPreds(t) \bullet DFlatten_x$ is well defined.

Finally, from case C1 and C2 we conclude that $\forall D : \text{DaMod0} \bullet \forall t : DObjs \bullet DFlatten_t$ is well defined.

□

To prove
Property *DProp8.2* that the auxiliary feature is well defined. That is, to prove that
$\forall D : \text{DaMod0} \bullet \forall t : DObjs \bullet \forall r : (t \cap \text{Roles}) \bullet$
The definition of $DFlatPart_{t,\,r}$ specifies exactly one member of $DFlatCart(t) \rightarrow DFlatCart(DConn(r))$

Assume that $D : \text{DaMod0}$ and $t : DObjs$.

Recall the definition : $\forall r : t \cap \text{Roles} \bullet \forall T : DFlatCart(t) \bullet$
$DFlatPart_t(r, T) = T'$ where $T' : DFlatCart(DConn(r))$ and
$\forall p : DFlatIndex(DConn(r)) \bullet T'Val(p) = TVal(p + \{r\})$

The proof is by cases, and there are two cases to consider.

C1 Case $t \subseteq \text{Entities}$
Then $t \cap \text{Roles} = \varnothing$ and the definition domain of $DFlatPart_t$ is empty. The property for this case is vacuously true.

We conclude that $DFlatPart_t$ is well defined. (Though this case is not used when unflattening).

C2 Case $t \subseteq \text{Roles}$
Then $t \cap \text{Roles} \neq \varnothing$ and by *DProp7.13* $DFlatCart(t) \neq \varnothing$.

We wish to prove that for each role $r : t$ and each tuple of $DFlatCart(t)$ there is exactly one tuple of $DFlatCart(DConn(r))$ that obeys the definition. For this to be so the definition must assign exactly one value to each index belonging to $DFlatIndex(DConn(r))$, and the value must belong to the domain indicated by $DFlatDomf_{DConn(r)}$.

Assume that $r : t$ and $T : DFlatCart(t)$. Recall that every fact-style tuple has a value function, and that the function is defined at each of the tuple's indexes. Recall also that the index set of T is $DFlatIndex(t)$.

There are two cases to consider.

C2.C1 Case $DConn(r) \subseteq \text{Entities}$
Then by *DProp7.1* $DFlatIndex(DConn(r)) = \varnothing$ so $DFlatCart(DConn(r)) = \Phi$ where Φ is the nullary cartesian product $\{\phi\}$. The only total function from $DFlatCart(t)$ to Φ, which maps T to ϕ, obeys the definition, vacuously in part.

We conclude that $DFlatPart_{t,\, r}$ is well defined. (Though this case is also not used when unflattening).

C2.C2 Case $DConn(r) \subseteq \text{Roles}$

Then by *DProp7.1* $DFlatIndex(DConn(r)) \neq \emptyset$. Assume that $p : DFlatIndex(DConn(r))$.

From the definition of *DFlatIndex* we have $p + \{r\} \in DFlatIndex(t)$.

By *DProp7.6* there is exactly one member, say s, of p such that $DConn(s) \subseteq \text{Entities}$. $s \in p + \{r\}$. By their definitions $DFlatDomf_{DConn(r)}(p) = DConn(s) = DFlatDomf_t(p + \{r\})$. By the definition of *DFlatPart* p is assigned the value $TVal(p + \{r\}) \in DFlatDomf_t(p + \{r\}) = DFlatDomf_{DConn(r)}(p)$.

We conclude that each $p : DFlatIndex(DConn(r))$ is assigned exactly one value of the correct domain, and hence we conclude that $DFlatPart_{t,\, r}$ is well defined.

Continuing case C2, from case C2.C1 and C2.C2 we conclude that $DFlatPart_{t,\, r}$ is well defined for each $r : t$, and hence we conclude that $DFlatPart_t$ is well defined.

Finally, from case C1 and C2 we conclude that

$\forall D : \text{DaMod0} \bullet \forall t : DObjs \bullet DFlatPart_t$ is well defined.

□

To prove

Property *DProp8.3* that unflattening is well defined. That is, to prove that

$\forall D : \text{DaMod0} \bullet \forall t : DObjs \bullet$

The definition of $DUnflatten_t$ specifies exactly one member of $DFlatCart(t) \rightarrow DCart(t)$.

Assume that $D : \text{DaMod0}$ and $t : DObjs$.

Recall the definition : $\forall T' : DFlatCart(t) \bullet$

$DUnflatten_t(T') = T$ where $T : DCart(t)$ and

$\forall r : (t \cap \text{Roles}) \bullet$

$[\, DConn(r) \subseteq \text{Entities} \Rightarrow TVal(r) = T'Val(\, \{r\}\,)\,] \wedge$

$[\, DConn(r) \subseteq \text{Roles} \Rightarrow$

$TVal(r) = DUnflatten_{DConn(r)}(DFlatPart_t(r, T'))\,]$

The proof is by Well Founded induction on *DObjs*. It is much the same as the proof for flattening except that many details have already been attended to in the proof for *DFlatPart*. There are two cases to consider.

C1 Case $t \subseteq \text{Entities}$

Then by *DProp7.1* $DFlatIndex(t) = \emptyset$ so $DCart(t) = \Phi = DFlatCart(t)$ where Φ is the nullary cartesian product $\{\phi\}$. The only total function from Φ to Φ, which maps ϕ to ϕ, obeys the definition, vacuously in part.

We conclude that $DUnflatten_t$ is well defined.

C2 Case $t \subseteq \text{Roles}$

Then by *DProp7.1* $DFlatIndex(t) \neq \emptyset$ and by *DProp7.13* $DFlatCart(t) \neq \emptyset$.

We wish to prove that for each tuple of *DFlatCart*(t) there is exactly one tuple of *DCart*(t) that obeys the definition. For this to be so the definition must assign exactly one value to each index and the value must belong to the domain indicated by $DDomf_t$. Recall that the index set of each tuple of *DCart*(t) is t itself.

Assume that T' : *DFlatCart*(t) and $r : t$. Recall that every fact-style tuple has a value function, and that the function is defined at each of the tuple's indexes. Recall also that the index set of T' is *DFlatIndex*(t).

Consider two cases.

C2.C1 Case $DConn(r) \subseteq \text{Entities}$

Then from the definition of *DFlatIndex* we have $\{r\} \in DFlatIndex(t)$ and from their definitions $DDomf_t(r) = DConn(r) = DFlatDomf_t(\ \{r\}\ )$. r is assigned the value $T'Val(\ \{r\}\ ) \in DFlatDomf_t(\ \{r\}\ ) = DDomf_t(r)$.

We conclude that r is assigned exactly one value and it belongs to the correct domain.

C2.C2 Case $DConn(r) \subseteq \text{Roles}$

Then from its definition $DDomf_t(r) = DCart(DConn(r))$.

We have $r \in t \cap \text{Roles}$ and $T' \in DFlatCart(t)$ so by *DProp8.2* $DFlatPart_t(r, T')$ is a uniquely determined member of $DFlatCart(DConn(r))$.

Assume for each $x : DPreds(t)$ that $DUnflatten_x$ is a uniquely determined total function from *DFlatCart*(x) to *DCart*(x). Then in particular this is so when $x = DConn(r)$.

r is assigned the value $DUnflatten_{DConn(r)}(DFlatPart_t(r, T')) \in DCart(DConn(r)) = DDomf_t(r)$.

We conclude that r is assigned exactly one value and it belongs to the correct domain.

Continuing case C2, from case C2.C1 and C2.C2 we conclude that each r is assigned exactly one value of the correct domain, and hence we conclude that $DUnflatten_t$ is well defined, provided that $\forall x : DPreds(t) \bullet DUnflatten_x$ is well defined.

Finally, from case C1 and C2 we conclude that

$\forall D : \text{DaMod0} \bullet \forall t : DObjs \bullet DUnflatten_t$ is well defined.

□

To prove

Property *DProp8.4* concerning the auxiliary feature. That is, to prove that

$$\forall D : \text{DaMod0} \bullet \forall t : DObjs \bullet \forall T : DCart(t) \bullet \forall r : t \cap \text{Roles} \bullet$$
$$DConn(r) \subseteq \text{Roles} \Rightarrow DFlatPart_t(r, DFlatten_t(T)) = DFlatten_{DConn(r)}(TVal(r))$$

This property is needed for the proof of *DProp8.5* (flattening is undone by unflattening).

Assume that D : DaMod0, $t : DObjs$, and $T : DCart(t)$. If $t \subseteq \text{Entities}$ then the property is vacuously true.

Assume that $t \subseteq$ Roles and $r : t$. If $DConn(r) \subseteq$ Entities then again the property is vacuously true.

Assume that $DConn(r) \subseteq$ Roles.

Recall the definition of *DFlatten* :

$\forall t : DObjs \bullet \forall T : DCart(t) \bullet$
$DFlatten_t(T) = T'$ where $T' : DFlatCart(t)$ and
$\forall p : DFlatIndex(t) \bullet \forall r : (p \cap t) \bullet$
$\quad [\, p = \{r\} \Rightarrow T'Val(p) = TVal(r)\,] \wedge$
$\quad [\, p \neq \{r\} \Rightarrow T'Val(p) = DFlatten_{DConn(r)}(TVal(r))Val(\, p - \{r\})\,]$

and of *DFlatPart* :

$\forall t : DObjs \bullet \forall r : t \cap \text{Roles} \bullet \forall T : DFlatCart(t) \bullet$
$DFlatPart_t(r, T) = T'$ where $T' : DFlatCart(DConn(r))$ and
$\forall p : DFlatIndex(DConn(r)) \bullet T'Val(p) = TVal(p + \{r\})$.

First we should confirm that the functions *DFlatten* and *DFlatPart* have been used properly. $DFlatten_t$ requires and is given a member of $DCart(t)$; $DFlatPart_t$ requires and is given a member of $t \cap$ Roles and a member of $DFlatCart(t)$; *TVal* requires and is given a member of t; $DFlatten_{DConn(r)}$ requires and from the definition of $DDomf_t$ is given a member of $DCart(DConn(r))$.

Now we wish to prove the equality. Each side of the equation is a tuple belonging to $DFlatCart(DConn(r))$. The tuples are equal if they have the same value at each index.

Assume $p : DFlatIndex(DConn(r))$.

From the definition of *DFlatPart* we have
$DFlatPart_t(r, DFlatten_t(T))Val(p) = DFlatten_t(T)Val(p + \{r\})$.
By *DProp7.6* $p \neq \varnothing$ and also $p \neq \{r\}$ so $p + \{r\} \neq \{r\}$. Then from the definition of *DFlatten* we have
$DFlatten_t(T)Val(p + \{r\}) = DFlatten_{DConn(r)}(TVal(r))Val(\,(p + \{r\}) - \{r\}) =$
$DFlatten_{DConn(r)}(TVal(r))Val(p)$.

We conclude that $DFlatPart_t(r, DFlatten_t(T)) = DFlatten_{DConn(r)}(TVal(r))$ and hence that *DProp8.4* is true.

□

To prove
Property *DProp8.5* that flattening is undone by unflattening. That is, to prove that
$\forall D : \text{DaMod0} \bullet \forall t : DObjs \bullet \forall T : DCart(t) \bullet DUnflatten_t(DFlatten_t(T)) = T$

Assume that D : DaMod0 and $t : DObjs$.

Recall the definition of *DFlatten* :
$\forall T : DCart(t) \bullet$
$DFlatten_t(T) = T'$ where $T' : DFlatCart(t)$ and
$\forall p : DFlatIndex(t) \bullet \forall r : (p \cap t) \bullet$
$\quad [\, p = \{r\} \Rightarrow T'Val(p) = TVal(r)\,] \wedge$
$\quad [\, p \neq \{r\} \Rightarrow T'Val(p) = DFlatten_{DConn(r)}(TVal(r))Val(\, p - \{r\})\,]$

and of *DUnflatten* :

$\forall T' : DFlatCart(t) \bullet$

$DUnflatten_t(T') = T$ where $T : DCart(t)$ and

$\forall r : (t \cap \text{Roles}) \bullet$

$[\, DConn(r) \subseteq \text{Entities} \Rightarrow TVal(r) = T'Val(\, \{r\}\,)\,] \wedge$

$[\, DConn(r) \subseteq \text{Roles} \Rightarrow$

$TVal(r) = DUnflatten_{DConn(r)}(DFlatPart_t(r, T'))\,]$.

The proof is by Well Founded induction on *DObjs*. There are two cases to consider.

C1 Case $t \subseteq$ Entities

Then by *DProp7.1* $DFlatIndex(t) = \varnothing$ so $DCart(t) = \Phi = DFlatCart(t)$ where Φ is the nullary cartesian product $\{\phi\}$. $DFlatten_t$ maps ϕ to ϕ and so does $DUnflatten_t$.

We conclude that $\forall T : DCart(t) \bullet DUnflatten_t(DFlatten_t(T)) = T$.

C2 Case $t \subseteq$ Roles

Assume that $T : DCart(t)$ and let $T' =_d DUnflatten_t(DFlatten_t(T))$.

We wish to prove that $T' = T$. These tuples are equal if they have the same value at each index. Recall that their index set is t.

Assume that $r : t$. There are two cases to consider.

C2.C1 Case $DConn(r) \subseteq$ Entities

Then from the definition of *DUnflatten* we have $T'Val(r) = DFlatten_t(T)Val(\, \{r\}\,)$, and from the definition of *DFlatten* we have $DFlatten_t(T)Val(\, \{r\}\,) = TVal(r)$.

Thus $T'Val(r) = TVal(r)$.

C2.C2 Case $DConn(r) \subseteq$ Roles

Then from the definition of *DUnflatten* we have $T'Val(r) = DUnflatten_{DConn(r)}(DFlatPart_t(r, DFlatten_t(T)))$, and by *DProp8.4* this is $DUnflatten_{DConn(r)}(DFlatten_{DConn(r)}(TVal(r)))$.

Assume for each $x : DPreds(t)$ that

$\forall T : DCart(x) \bullet DUnflatten_x(DFlatten_x(T)) = T$,

in particular when $x = DConn(r)$.

Then $DUnflatten_{DConn(r)}(DFlatten_{DConn(r)}(TVal(r))) = TVal(r)$.

Thus $T'Val(r) = TVal(r)$.

Continuing case C2, from case C2.C1 and C2.C2 we conclude that

$\forall T : DCart(t) \bullet DUnflatten_t(DFlatten_t(T)) = T$

provided that

$\forall x : DPreds(t) \bullet \forall T : DCart(x) \bullet DUnflatten_x(DFlatten_x(T)) = T$.

Finally, from case C1 and C2 we conclude that

$\forall D : \text{DaMod0} \bullet \forall t : DObjs \bullet$

$\forall T : DCart(t) \bullet DUnflatten_t(DFlatten_t(T)) = T$.

□

To prove

Property *DProp8.6* that unflattening is undone by flattening. That is, to prove that

$\forall D$: DaMod0 $\bullet$ $\forall t : DObjs$ $\bullet$ $\forall T' : DFlatCart(t)$ $\bullet$
$DFlatten_t(DUnflatten_t(T')) = T'$

Assume that D : DaMod0 and $t : DObjs$.

Remember that the definitions of *DFlatten* and *DUnflatten* were repeated at the beginning of the previous proof. The definition of *DFlatPart* is repeated here :

$\forall t : DObjs$ $\bullet$ $\forall r : t \cap$ Roles $\bullet$ $\forall T : DFlatCart(t)$ $\bullet$
$DFlatPart_t(r, T) = T'$ where $T' : DFlatCart(DConn(r))$ and
$\forall p : DFlatIndex(DConn(r))$ $\bullet$ $T'Val(p) = TVal(p + \{r\})$.

The proof is by Well Founded induction on *DObjs*. There are two cases to consider.

C1 Case $t \subseteq$ Entities

Then by *DProp7.1* $DFlatIndex(t) = \varnothing$ so $DFlatCart(t) = \Phi = DCart(t)$ where Φ is the nullary cartesian product $\{\phi\}$. $DUnflatten_t$ maps ϕ to ϕ and so does $DFlatten_t$.

We conclude that $\forall T' : DFlatCart(t)$ $\bullet$ $DFlatten_t(DUnflatten_t(T')) = T'$.

C2 Case $t \subseteq$ Roles

Assume that $T' : DFlatCart(t)$ and let $T'' =_d DFlatten_t(DUnflatten_t(T'))$.

We wish to prove that $T'' = T'$. These tuples are equal if they have the same value at each index. Recall that their index set is $DFlatIndex(t)$.

Assume that $p : DFlatIndex(t)$. By *DProp7.5* $p \cap t$ is a singleton set, say $\{r\}$. Either $p = \{r\}$ or it does not. Consider these two cases.

C2.C1 Case $p = \{r\}$

Then from the definition of *DFlatten* we have
$T''Val(\ \{r\}\ ) = DUnflatten_t(T')Val(r)$. By *DProp7.6* $DConn(r) \subseteq$ Entities so from the definition of *DUnflatten* we have
$DUnflatten_t(T')Val(r) = T'Val(\ \{r\}\ )$.

Thus $T''Val(p) = T'Val(p)$.

C2.C2 Case $p \neq \{r\}$

Then from the definition of *DFlatten* we have
$T''Val(p) = DFlatten_{DConn(r)}(DUnflatten_t(T')Val(r))Val(\ p - \{r\})$. From the definition of *DFlatIndex* $DConn(r) \subseteq$ Roles (for if not $p = \{r\}$), so from the definition of *DUnflatten* this is
$DFlatten_{DConn(r)}(DUnflatten_{DConn(r)}(DFlatPart_t(r, T')))Val(\ p - \{r\})$

Assume for each $x : DPreds(t)$ that

$\forall T' : DFlatCart(x)$ $\bullet$ $DFlatten_x(DUnflatten_x(T')) = T'$,

in particular when $x = DConn(r)$.

Then
$DFlatten_{DConn(r)}(DUnflatten_{DConn(r)}(DFlatPart_t(r, T')))Val(\ p - \{r\}) =$
$DFlatPart_t(r, T')Val(\ p - \{r\})$.

From the definition of *DFlatPart* we have
$DFlatPart_t(r, T')Val(\ p - \{r\}) = T'Val(\ (p - \{r\}) + \{r\}) = T'Val(p)$.

Thus $T''Val(p) = T'Val(p)$.

Continuing case C2, from case C2.C1 and C2.C2 we conclude that
$\forall T' : DFlatCart(t) \bullet DFlatten_t(DUnflatten_t(T')) = T'$
provided that
$\forall x : DPreds(t) \bullet \forall T' : DFlatCart(x) \bullet DFlatten_x(DUnflatten_x(T')) = T'$,

Finally, from case C1 and C2 we conclude that
$\forall D : \mathsf{DaMod0} \bullet \forall t : DObjs \bullet$
$\forall T' : DFlatCart(t) \bullet DFlatten_t(DUnflatten_t(T')) = T'$.

□

To prove
(At last !) Property *DProp8.7* that flattening and unflattening are described by bijections that are inverses of each other. That is, to prove that
$\forall D : \mathsf{DaMod0} \bullet \forall t : DObjs \bullet$
$\mathsf{IsBijection}(DFlatten_t) \wedge \mathsf{IsBijection}(DUnflatten_t) \wedge DUnflatten_t = DFlatten_t^{-1}$

We start by proving a general result about functions. The result will then be applied to the flattening and unflattening functions.

Assume that we are given any two sets X, Y and two functions
$f : X \to Y$, $g : Y \to X$ with the properties that
$\forall x : X \bullet g(f(x)) = x$ and $\forall y : Y \bullet f(g(y)) = y$.
We wish to prove that f and g are bijections and are inverses of each other.

For any $y : Y$ we have $f(g(y)) = y$ so $\mathsf{Ran}(f) = Y$; f is surjective.

For any $x, x' : X$ if $f(x) = f(x')$ then we have $x = g(f(x)) = g(f(x')) = x'$; f is injective.

For any $x : X$ we have $g(f(x)) = x$ so $\mathsf{Ran}(g) = X$; g is surjective.

For any $y, y' : Y$ if $g(y) = g(y')$ then we have $y = f(g(y)) = f(g(y')) = y'$; g is injective.

Both f and g are injective and surjective, and we are given that they are total and functional. We conclude that f and g are bijections. We are given that they are inverses of each other. That is, that $g = f^{-1}$ and $f = g^{-1}$.

Now we apply this result.

Assume that $D : \mathsf{DaMod0}$ and $t : DObjs$.

Then
$DFlatten_t : DCart(t) \to DFlatCart(t)$ by definition;
$DUnflatten_t : DFlatCart(t) \to DCart(t)$ by definition;
$\forall T : DCart(t) \bullet DUnflatten_t(DFlatten_t(T)) = T$ by *DProp8.5*; and
$\forall T' : DFlatCart(t) \bullet DFlatten_t(DUnflatten_t(T')) = T'$ by *DProp8.6*.

We conclude from the general result that $DFlatten_t$ and $DUnflatten_t$ are bijections and that $DUnflatten_t = DFlatten_t^{-1}$, and hence that *DProp8.7* is true.

□

Finally we define the equivalence relation FlatEq and state and prove some of its properties.

<u>FlatEq</u>

Predicate that is true of two objects of two members of DaMod0 iff the Fact Types identified by the objects are encoded versions of each other

Given any $D1, D2$: DaMod0, $t1 : D1Objs$, $t2 : D2Objs$

then

$$\begin{aligned}
&\text{FlatEq}((D1, t1), (D2, t2)) \Leftrightarrow_{\text{d}}\\
&\quad \exists B : D1FlatIndex(t1) \leftrightarrow D2FlatIndex(t2) \bullet\\
&\qquad \text{IsBijection}(B) \wedge\\
&\qquad \forall p : D1FlatIndex(t1) \bullet D1FlatDomf_{t1}(p) = D2FlatDomf_{t2}(B(p))
\end{aligned}$$

<u>FlatEq Properties 1</u>

Some properties of the relation FlatEq

FlatEqProp1.1 .:

$$\begin{aligned}
&\forall D1, D2 : \text{DaMod0} \bullet \forall t1 : D1Objs \bullet \forall t2 : D2Objs \bullet\\
&\quad t1 \subseteq \text{Entities} \Rightarrow [\, t2 \subseteq \text{Entities} \Leftrightarrow \text{FlatEq}((D1, t1), (D2, t2)) \,]
\end{aligned}$$

FlatEqProp1.3 .: There is an alternative definition of FlatEq

$$\begin{aligned}
&\forall D1, D2 : \text{DaMod0} \bullet \forall t1 : D1Objs \bullet \forall t2 : D2Objs \bullet\\
&\quad \text{FlatEq}((D1, t1), (D2, t2))\\
&\quad \Leftrightarrow\\
&\quad \forall E \subseteq_{\text{d}} \text{Entities} \bullet\\
&\qquad \text{Num}(\{ p : D1FlatIndex(t1) \mid D1FlatDomf_{t1}(p) = E \}) =\\
&\qquad \text{Num}(\{ p : D2FlatIndex(t2) \mid D2FlatDomf_{t2}(p) = E \})
\end{aligned}$$

FlatEqProp1.4 .: There is a natural bijection associated with FlatEq

$$\begin{aligned}
&\forall D1, D2 : \text{DaMod0} \bullet \forall t1 : D1Objs \bullet \forall t2 : D2Objs \bullet\\
&\forall B : D1FlatIndex(t1) \leftrightarrow D2FlatIndex(t2) \bullet\\
&\quad [\, \text{IsBijection}(B) \wedge\\
&\quad\ \ \forall p : D1FlatIndex(t1) \bullet D1FlatDomf_{t1}(p) = D2FlatDomf_{t2}(B(p)) \,]\\
&\quad \Rightarrow\\
&\quad \exists 1\ B' : D1FlatCart(t1) \leftrightarrow D2FlatCart(t2) \bullet\\
&\qquad \text{IsBijection}(B') \wedge\\
&\qquad \forall T : D1FlatCart(t1) \bullet \forall p : D1FlatIndex(t1) \bullet TVal(p) = B'(T)Val(B(p))
\end{aligned}$$

FlatEqProp1.5 .: Given sufficient spare roles, any Fact Type can be represented by a Fact Type of rank 1

$\forall D$: DaMod0 $\bullet$ $\forall t : DObjs$ $\bullet$
$t \subseteq$ Roles $\wedge$ [$\exists F : DFlatIndex(t) \rightarrow$ (Roles - $DRoles$) $\bullet$ IsInjection(F)]
$\Rightarrow$
[$\exists D'$: DaMod0 $\bullet$ D AddedRo D' $\wedge$
$\forall t' : (D'Objs - DObjs)$ $\bullet$ $D'Rank(t') = 1$ $\wedge$ FlatEq((D, t), (D', t'))]

FlatEqProp1.1 is immediate from *DProp7.1* and the definition of FlatEq.

To prove
Property FlatEqProp1.3 that there is an alternative definition of FlatEq. That is, to prove that

$\forall D1, D2$: DaMod0 $\bullet$ $\forall t1 : D1Objs$ $\bullet$ $\forall t2 : D2Objs$ $\bullet$
FlatEq(($D1$, $t1$), ($D2$, $t2$))
$\Leftrightarrow$
$\forall E \subseteq_d$ Entities $\bullet$
Num($\{ p : D1FlatIndex(t1) \mid D1FlatDomf_{t1}(p) = E \}$) =
Num($\{ p : D2FlatIndex(t2) \mid D2FlatDomf_{t2}(p) = E \}$)

Let $\alpha =_{SYM}$ FlatEq(($D1$, $t1$), ($D2$, $t2$))

Let $\beta =_{SYM}$ $\forall E \subseteq_d$ Entities $\bullet$
Num($\{ p : D1FlatIndex(t1) \mid D1FlatDomf_{t1}(p) = E \}$) =
Num($\{ p : D2FlatIndex(t2) \mid D2FlatDomf_{t2}(p) = E \}$)

Assume that $D1, D2$: DaMod0, $t1 : D1Objs$, and $t2 : D2Objs$.

We wish to prove that $\alpha \Leftrightarrow \beta$. The forward and reverse implications are proved separately. We will use the fact that given any sets x, y then Num(x) = Num(y) iff there is a bijection from x to y.

F $\alpha \Rightarrow \beta$

Assume that α is true. Then there is a bijection from $D1FlatIndex(t1)$ to $D2FlatIndex(t2)$, say B, such that
$\forall p : D1FlatIndex(t1)$ $\bullet$ $D1FlatDomf_{t1}(p) = D2FlatDomf_{t2}(B(p))$.

Assume that $E \subseteq$ Entities.

To make this proof easier to read
Let $a =_d \{ p : D1FlatIndex(t1) \mid D1FlatDomf_{t1}(p) = E \}$, and
Let $b =_d \{ p : D2FlatIndex(t2) \mid D2FlatDomf_{t2}(p) = E \}$.

We wish to construct a bijection from a to b.

For any $p : D1FlatIndex(t1)$, from the definitions of a, b, and B, we have
$p \in a \Leftrightarrow D1FlatDomf_{t1}(p) = E \Leftrightarrow D2FlatDomf_{t2}(B(p)) = E \Leftrightarrow B(p) \in b$.

First, we have $p \in a \Rightarrow B(p) \in b$ from which we conclude that $B[[a]] \subseteq b$.

Second, for any $p' : D2FlatIndex(t2)$ we have $B^{-1}(p') \in D1FlatIndex(t1)$. Substituting $B^{-1}(p')$ for p we have $B^{-1}(p') \in a \Leftrightarrow B(B^{-1}(p')) \in b$ from which we conclude that $p' \in b \Rightarrow p' \in B[[a]]$ and hence that $b \subseteq B[[a]]$.

Thus $B[[a]] = b$. If we restrict the bijection B to the domain a and codomain $B[[a]]$ we have our bijection from a to b. It follows that $\mathsf{Num}(a) = \mathsf{Num}(b)$.

We conclude that β is true and hence that $\alpha \Rightarrow \beta$.

R $\beta \Rightarrow \alpha$

Assume that β is true.

We wish to construct a bijection from *D1FlatIndex*(*t1*) to *D2FlatIndex*(*t2*).

Choose a family $B =_d (B_E \mid E \subseteq_d \mathsf{Entities})$ where for each $E \subseteq_d \mathsf{Entities}$ B_E is a bijection from $\{ p : D1FlatIndex(t1) \mid D1FlatDomf_{t1}(p) = E \}$ to $\{ p : D2FlatIndex(t2) \mid D2FlatDomf_{t2}(p) = E \}$. From β we can be sure that there is at least one such bijection to choose from for each E. However, as we might be making an infinite number of choices we should make sure that there is a set obeying the description of B.

It is only when $E \in \mathsf{Ran}(D1FlatDomf_{t1})$ that $\mathsf{Num}(\{ p : D1FlatIndex(t1) \mid D1FlatDomf_{t1}(p) = E \})$ can be non-zero. When it is zero the bijection B_E is uniquely determined. Now $\mathsf{Ran}(D1FlatDomf_{t1}) \subseteq D1Objs$ which is finite by *DProp1.1* (in Section 4.3.2). Thus there are only a finite number of choices to be made; we can be sure that the family B exists without invoking the Axiom of Choice.

Now form the relation $B' : D1FlatIndex(t1) \leftrightarrow D2FlatIndex(t2)$ whose graph is defined to be $B'Gr =_d \bigcup \{ B_E Gr \mid E \subseteq_d \mathsf{Entities} \}$. We wish to prove that B' is a bijection.

Assume that $p : D1FlatIndex(t1)$. $D1FlatDomf_{t1}$ is a function and by *DProp7.9* $p \in \mathsf{Def}(D1FlatDomf_{t1})$. By its definition $D1FlatDomf_{t1}(p) \subseteq \mathsf{Entities}$. Thus p is a member of the set $\{ p : D1FlatIndex(t1) \mid D1FlatDomf_{t1}(p) = E \}$ for exactly one $E \subseteq_d \mathsf{Entities}$. By the definition of B there is exactly one $p' : D2FlatIndex(t2)$ such that $B_E(p) = p'$. Thus $B'Gr$ contains exactly one couple of the form $\langle p, p'' \rangle$. We conclude that B' is total and functional.

By the same reasoning applied to $D2FlatIndex(t2)$ using B_E^{-1} we conclude that B' is surjective and injective, and hence that B' is a bijection.

Finally, by the constructions of B and B' we have for any $p : D1FlatIndex(t1)$ that
$D1FlatDomf_{t1}(p) = D2FlatDomf_{t2}(B_{D1FlatDomf(t1)(p)}(p)) = D2FlatDomf_{t2}(B'(p))$.
We conclude that α is true and hence that $\beta \Rightarrow \alpha$.

From F and R we conclude that $\alpha \Leftrightarrow \beta$ and hence that FlatEqProp1.3 is true.

□

To prove
Property FlatEqProp1.4 that there is a natural bijection associated with FlatEq. That is, to prove that

$\forall D1, D2 : \mathrm{DaMod0} \bullet \forall t1 : D1Objs \bullet \forall t2 : D2Objs \bullet$
$\forall B : D1FlatIndex(t1) \leftrightarrow D2FlatIndex(t2) \bullet$
$\quad [\ \mathrm{IsBijection}(B) \wedge$
$\quad \forall p : D1FlatIndex(t1) \bullet D1FlatDomf_{t1}(p) = D2FlatDomf_{t2}(B(p))\ ]$
$\quad \Rightarrow$
$\quad \exists 1\ B' : D1FlatCart(t1) \leftrightarrow D2FlatCart(t2) \bullet$
$\qquad \mathrm{IsBijection}(B') \wedge$
$\qquad \forall T : D1FlatCart(t1) \bullet \forall p : D1FlatIndex(t1) \bullet TVal(p) = B'(T)Val(B(p))$

Assume that $D1, D2$: DaMod0, $t1 : D1Objs$, and $t2 : D2Objs$.

Assume that $B : D1FlatIndex(t1) \to D2FlatIndex(t2)$, that B is a bijection, and
$\forall p : D1FlatIndex(t1) \bullet D1FlatDomf_{t1}(p) = D2FlatDomf_{t2}(B(p))$.

We will use the general property CartProdProp2.1 (in Section 5.4.2) concerning the conversion of a cartesian product's index set and domains.

To make this proof easier to read
Let $P =_{\mathrm{d}} D1FlatIndex(t1)$.

First we construct a family of bijections to describe the transformation of $D1FlatDomf_{t1}$ into $D2FlatDomf_{t2}$.
Let $C =_{\mathrm{d}} (\ C_p \mid p : P \cup \{P\}\ )$ such that
$C_P =_{\mathrm{d}} B$ and
$\forall p : P \bullet C_p =_{\mathrm{d}} \mathrm{Id}(D1FlatDomf_{t1}(p))$. (Identity function)

This translates the domain function $D1FlatDomf_{t1}$ into the set
$\{ \langle C_P(p), C_p[[\ D1FlatDomf_{t1}(p)\ ]]\rangle \mid p : P \}$, which is indeed $D2FlatDomf_{t2}$.

By CartProdProp2.1 there is exactly one bijection, say B', from
$\mathrm{CartProd}(D1FlatDomf_{t1})$ to
$\mathrm{CartProd}(\{ \langle C_P(p), C_p[[\ D1FlatDomf_{t1}(p)\ ]]\rangle \mid p : P \})$
such that
$\forall T : \mathrm{CartProd}(D1FlatDomf_{t1}) \bullet \forall p : P \bullet C_p(TVal(p)) = B'(T)Val(C_P(p))$,
which is to say from
$D1FlatCart(t1)$ to $D2FlatCart(t2)$ such that
$\forall T : D1FlatCart(t1) \bullet \forall p : D1FlatIndex(t1) \bullet TVal(p) = B'(T)Val(B(p))$.

We conclude that FlatEqProp1.4 is true.

□

To prove
Property FlatEqProp1.5 that, given sufficient spare roles, any Fact Type can be represented by a Fact Type of rank 1. That is, to prove that

$\forall D : \mathrm{DaMod0} \bullet \forall t : DObjs \bullet$
$t \subseteq \mathrm{Roles} \wedge [\ \exists F : DFlatIndex(t) \to (\mathrm{Roles} - DRoles) \bullet \mathrm{IsInjection}(F)\ ]$
$\Rightarrow$
$[\ \exists D' : \mathrm{DaMod0} \bullet D \ \mathrm{AddedRo}\ D' \wedge$
$\quad \forall t' : (D'Objs - DObjs) \bullet D'Rank(t') = 1 \wedge \mathrm{FlatEq}((D, t), (D', t'))\ ]$

Assume that D : DaMod0 and t : *DObjs*.

Assume that $t \subseteq$ Roles.

Assume that F : *DFlatIndex*(t) $\rightarrow$ (Roles - *DRoles*) and IsInjection(F).

First define some more variables.
Let $t' =_{\text{d}} F[[$ *DFlatIndex*(t) $]]$, the roles selected by F.
Notice that $t' \subseteq$ Roles and $t' \cap$ *DRoles* $= \emptyset$.

Let F' : Bijection be the result of restricting the codomain of F to t'.

Next we wish to construct an appropriate member of PreMod. Define the function C : Roles $+\!\!\rightarrow$ Objects where *CDef* $=_{\text{d}} t'$ and for each role $r : t'$ we have $C(r) =_{\text{d}}$ $DFlatDomf_t(F'^{-1}(r))$. Observe that *CDef* $\cap$ *DConnDef* $= \emptyset$ and for each $r : t'$ that $C(r) \subseteq$ Entities. Observe also that $t' \in$ Objects.

The desired member is D' : PreMod with objects *D'Objs* $=_{\text{d}}$ *DObjs* $\cup \{t'\}$ and with *D'Conn*'s graph *D'ConnGr* $=_{\text{d}}$ *DConnGr* $\cup$ *CGr*. By our choice of t' and C we have ensured that D' exists.

Next we wish to prove that D AddedRo D' and that $D' \in$ DaMod0. t' is a subset of Roles that is disjoint from each member of *DObjs*. $t \subseteq$ Roles so by *DProp7.1* *DFlatIndex*(t) is non-empty and by *DProp7.3* it is finite; thus t' is also non-empty and finite. By definition *D'Objs* = *DObjs* $\cup \{t'\}$. For each role $r : t'$ we have $r \in$ *D'ConnDef* and *D'Conn*(r) $= C(r) = DFlatDomf_t(F'^{-1}(r))$. By the definition of *DFlatDomf* there is some role s : *DRoles* such that $s \in F'^{-1}(r)$ with $DFlatDomf_t(F'^{-1}(r)) =$ *DConn*(s). Now *DConn*(s) $\in$ *DObjs* so *D'Conn*(r) $\in$ *DObjs*. Finally, for each r : (*D'ConnDef* - t') we have *D'Conn*(r) = *DConn*(r).

We conclude that D AddedRo D'. As $D \in$ DaMod0 then $D' \in$ DaMod0.

Next we wish to determine the rank of t'. For each $r : t'$ we have *D'Conn*(r) $= C(r)$ $\subseteq$ Entities so the rank of *D'Conn*(r) is zero. As the rank of each immediate predecessor of t' is zero then the rank of t' is one.

Finally, we wish to prove that t and t' are equivalent in the FlatEq sense. For each $r : t'$ we have *D'Conn*(r) $\subseteq$ Entities so *D'FlatIndex*(t') $= \{ \{r\} \mid r : t' \}$. Define the bijection $B : t' \rightarrow$ *D'FlatIndex*(t') with $B(r) =_{\text{d}} \{r\}$ for each $r : t'$.

Now *F'Cod* $= t' =$ *BDom* so we have a bijection $B \circ F'$ from *DFlatIndex*(t) to *D'FlatIndex*(t'). Moreover, for each p : *DFlatIndex*(t) we have
$DFlatDomf_t(p) = C(F'(p)) = D'Conn(F'(p)) =$
$D'FlatDomf_{t'}(\ \{F'(p)\}\ ) = D'FlatDomf_{t'}(B \circ F'(p))$.
We conclude that FlatEq((D, t), (D', t')).

From all these conclusions we conclude that FlatEqProp1.5 is true.

□

6 The completeness of the model

In Chapters 4 and 5 we specified and analysed a model of the core part of NIAM conceptual data models. However, the model is restricted to those data models that can be built in a particular way. Perhaps there are data models that deserve to be called well-formed but have been excluded from our model? In this chapter we will prove that the model does indeed include all well-formed NIAM data models, provided we accept two very reasonable assumptions.

This chapter starts with a discussion of considerations and the plan of attack (Section 6.1). This is followed by an overview of the proof (Section 6.2). The chapter ends with the mathematics (Section 6.3). The proof enables us to describe some limitations of the NIAM design technique. These are discussed briefly at the end of the overview.

6.1 Considerations and plan of attack

We saw in Chapter 4 that the primary purpose of the core part of a data model is to specify well defined Fact Types, alias cartesian products. In our model we have used a representation of tuples that ensures that any cartesian products that need to be distinguished will be different. Thus we can say that a data model specifies a set of cartesian products. We do not need to describe them as a family or a more complicated structure of cartesian products.

We also saw that our model has some limitations. For instance, each member of DaMod0 specifies a finite set of cartesian products, with finite arities. However, this is a natural limitation of the common NIAM notations. Thus the question is not whether DaMod0 has limitations but whether they are more restrictive than NIAM's.

The plan, then, is to model the class of all sets of cartesian products that NIAM data models can specify, then to confirm that each of these sets of cartesian products can be specified by some member of DaMod0. We will do this in stages. We will start by defining the most general class of all sets of cartesian products, then introduce NIAM restrictions step by step. At each step we will note some properties and definitions that will be used in the final proof.

6.2 Overview

We wish to study the properties of sets of cartesian products. We will start by reminding ourselves of the representation of tuples and cartesian products that is used here. Each fact-style tuple *T* : Tuple has three primary variable features : an index set *TI*, a domain function *TDomf*, and a value function *TVal*. As usual, a fact-style cartesian product is a set of all those tuples having a particular domain function. Two tuples belonging to different cartesian products can have the same index sets and value functions but will differ at least in their domain functions. Consequently, any two cartesian products are either equal or disjoint; there cannot be any partial overlap.

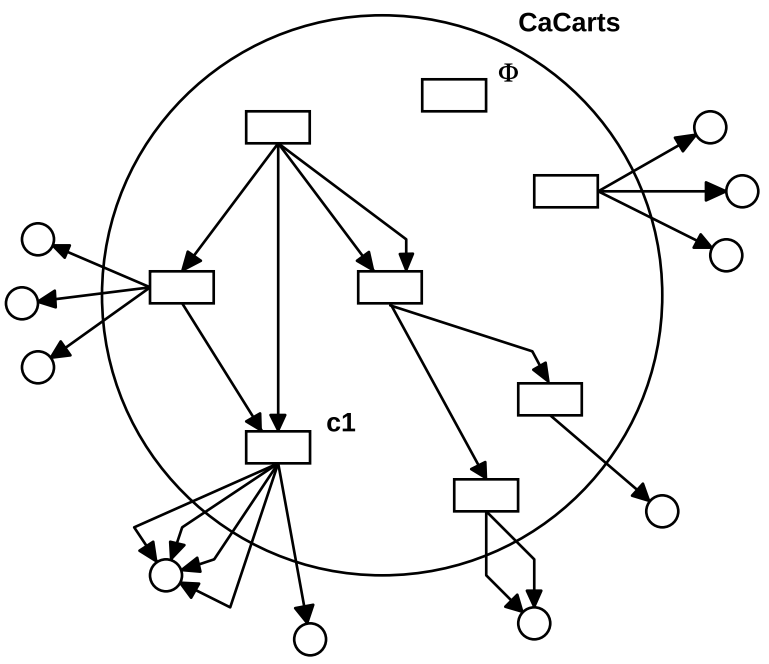
The first step in the proof of completeness is to define the class SetCart. Each member *C* of SetCart has one primary variable feature : a set, *CCarts*, of non-empty fact-style cartesian products. Note that *CCarts* can contain an infinite number of cartesian products and that their tuples can have infinite index sets.

If we inspect the tuples of a cartesian product belonging to *CCarts* we can obtain the index set and domain function its tuples have in common, and hence the domains that they use. Thus we can define some secondary features of *C* that will be used later on :

CInd :	For each *c* : *CCarts*, *CInd*(*c*) is the index set of *c*'s tuples;
CDomf :	For each *c* : *CCarts*, $CDomf_c$ is the domain function of *c*'s tuples;
CIndexes :	The set of all indexes occurring in *C*; i.e *i* is a member of *CIndexes* iff *i* is an index of some *c* : *CCarts*;
CDoms :	The set of all domains occurring in *C*; i.e *d* is a member of *CDoms* iff *d* is a domain of some *c* : *CCarts*, so $d = CDomf_c(i)$ for some index *i* of *c*.

For any *C* : SetCart the domain functions of the cartesian products induce a structure on *CCarts* that is best illustrated by a simple example. Figure 6.2.1 below is a picture of a particular member, Ca, of SetCart. The large circle represents CaCarts and the rectangles represent the cartesian products belonging to CaCarts. Each arrow represents the association of an index with a domain. Whether by accident or design each domain is either a member of CaCarts or it is not. If not, then it is shown as a small circle outside the big circle.

Figure 6.2.1 **A small member,** Ca**, of** SetCart



From the picture we can see that there are two more features of any *C* : SetCart that we can define :

CDomsBase :	The set of **base domains**, those domains of *C* that are not members of *CCarts*;
CIsDomOf :	Relation on *CCarts*; for any *c*, *c'* : *CCarts*, *c' CIsDomOf c* iff *c'* is a domain of *c*.

The small circles in Figure 6.2.1 represent the base domains of Ca. Notice that there is at least one cartesian product, c1 for instance, whose domains are all base domains. This illustrates a general property. For any *C* : SetCart, if *CCarts* is not empty then there will be a member whose domains are all base domains, vacuously so in the case of the nullary cartesian product Φ. A similar property is true of subsets of *CCarts* as well. If *X* is any non-empty subset of *CCarts* then there will be a member of *X* whose domains are base domains or, at least, not members of *X*. In other words, we have a fundamental property of sets of cartesian products : the relation *CIsDomOf* is a Well Founded relation.

We will use this property later on when we will want to choose a member of *CCarts* whose domains are either base domains or members of *CCarts* that were chosen earlier. The property ensures that there is always at least one to choose while any remain unchosen.

The property is proved in the details section but the proof relies on two assumptions which need to be highlighted here. The first assumption is that the elements of each tuple are held "inside" the tuple in some way. That is, each value forming an element of a tuple is either a member of the tuple, or a member of a member, or a member of a member of a member, … .

Although we defined the representation of tuples in considerable detail we left certain details open. For instance, each tuple is decreed to have three features, but we have not said how these features are to be held together. For most purposes it does not matter; any method that does the job will do. In addition, we would expect any useful statements about the properties of data models to be derivable whatever reasonable representation of tuples had been used.

Unfortunately, there are unusual representations that invalidate the assumption about membership. For instance, one particular tuple T with elements a and b could be defined to be represented by the set ∅. Provided no other tuple is represented by ∅ then this causes no confusion. Even worse, a could be defined to be ∅, alias T, so T is an element of itself. Now we cannot prove that *CIsDomOf* is Well Founded for every *C* : SetCart for the simple reason that it is not true.

However, the assumption about membership is true for the commonly used representations of tuples. It seems reasonable to say that perverse representations defined with special cases would be inappropriate in a general purpose design technique such as NIAM.

The second assumption is that the $\in$ relation is itself Well Founded. That is, the proof makes essential use of the Axiom of Foundation. The practical consequence of this is that no cartesian product is a domain of itself. More generally, there are no circular definitions; no cartesian product is defined in terms of itself. This seems appropriate : a data model should tell the implementers what is to be implemented, not pass on design problems that might not be solvable.

The next step in the proof of completeness is to restrict our attention to members of SetCart whose index sets have convenient properties. As indexes are arbitrary placeholders we can take any member of SetCart and replace its indexes so that the cartesian products have pairwise disjoint index sets. There is no loss of generality provided the result is still a member of SetCart. Rather than talk of replacing indexes we will simply restrict our attention from now on to the subclass SetCartDis of SetCart defined as follows. The members of SetCartDis are the members of SetCart whose cartesian products have pairwise disjoint index sets.

Each *C* : SetCartDis has two properties that will be used later on. First, each cartesian product belonging to *CCarts* is identified by its index set; no other member of *CCarts* has the same index set. Second, each index belonging to *CIndexes* is associated with exactly one domain, so *C* has an overall domain function. Thus we can define an additional secondary feature :

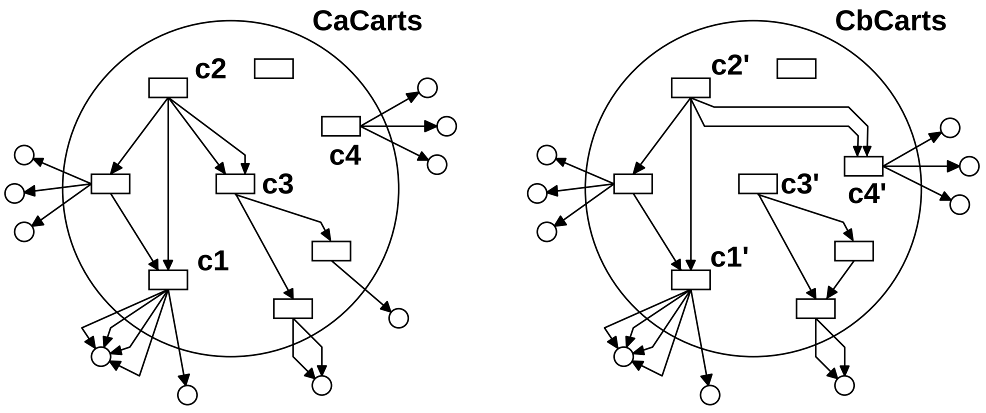
CInDom :	Total function from *CIndexes* to *CDoms*; $CInDom(i) =_{\mathrm{d}} d$ iff *d* is the domain assigned to the index *i* by some member of *CCarts*.

Notice that the members of SetCartDis are beginning to resemble the members of DaMod0 : cartesian products are identified by their index sets and there is an overall domain function assigning indexes to domains.

The next step in the proof of completeness is to extend the class SetCartDis by adding another primary variable feature. There is a deficiency in SetCartDis that must be remedied. Remember that the eventual purpose of SetCart and its derivatives is to model the sets of cartesian products defined by data models. The sets used in SetCartDis are arbitrary modelling elements. For any given data model it should not matter which modelling elements are used provided certain relationships are preserved. There will always be many members of SetCartDis able to model the given data model.

Recall that one of the operations defined in Chapter 5 was base conversion, alias isomorphism. The sets used as indexes and domain members were replaced to give a different, but equivalent, model. Let us do the same to Ca to give us Cb, as in Figure 6.2.2 below. Notice that one of the cartesian products, c2, has been changed to c2'. One of its domains, c1, has changed to c1', but another domain, c3, has changed to c4', not c3'! Is this proper?

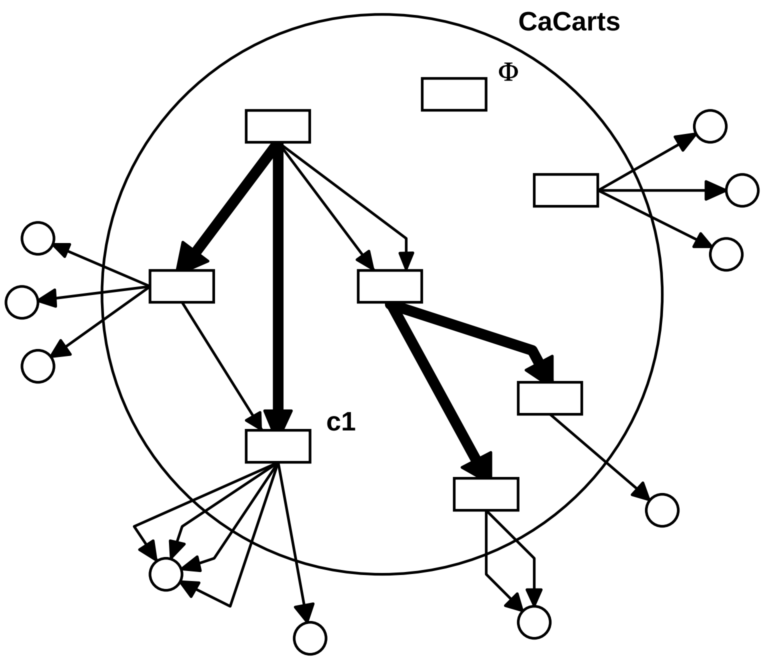
Figure 6.2.2 Base conversion of Ca **into** Cb



It can be proper. Suppose we have a data model with four Fact Types that are modelled by c1, c2, c3, and c4. Suppose the data model declares that the Fact Type modelled by c1 is a domain of the Fact Type modelled by c2. Then in any alternative model of the Fact Types c1' must be a domain of c2'. Suppose further that the data model declares that the domain used twice by c2 is an Entity Type containing exactly 64 entities, and that c3 and c4 each contain 64 tuples. As we range over SetCartDis selecting those members that model these Fact Types we will find that every set with 64 members is used to model the Entity Type. This includes every cartesian product with 64 tuples, such as c3 and c4'. The conversion of Ca into Cb is proper here.

Thus the transformation of Ca into Cb might or might not be proper, depending on the data model under consideration. How should we define legal transformations? In a picture we could highlight the arrows that must be preserved, as in Figure 6.2.3 below, but in the members of SetCart and SetCartDis there are no features that do the equivalent of highlighting some arrows but not others.

Figure 6.2.3 Ca with some arrows highlighted



We know that in a NIAM data model it is possible to declare that one of the specified cartesian products is a domain of another of the specified cartesian products. A legal base conversion must preserve such relationships. Although we will not define a base conversion operator here we will make sure it can be defined by adding another feature to the members of SetCartDis to give us the class SetCartNom. Each member *C* of SetCartNom has two primary variable features : a member of SetCartDis, with its primary feature *CCarts* and secondary features *CIndexes*, *CDoms*, etc, and the set *CNoms* of those indexes that are to be highlighted. *CNoms* is required to be a subset of *CIndexes*, and each index belonging to *CNoms* must be associated with a domain that is a member of *CCarts*.

We will say that the members of *CNoms* are the **nominated indexes** and that the domains they are associated with are **nominee domains**. A domain associated with an index that is not nominated is a base domain if it is not a member of *CCarts*, or is a member of *CCarts* "by accident" otherwise. In either case, we will say that it is a **basic domain**. Notice that it is possible for a domain such as c1 in Figure 6.2.3 to be both a basic domain and a nominee domain via different indexes.

Each *C* : SetCartNom has some additional secondary features :

CDomsNom :	The nominee domains; those members of *CDoms* assigned to an index belonging to *CNoms*;
CDomsBasic :	The basic domains; those members of *CDoms* assigned to an index not belonging to *CNoms*;
CIsDomOfNom :	Relation on *CCarts*; for any *c, c'* : *CCarts*, *c' CIsDomOfNom c* iff *c'* is a domain of *c* via a nominated index.

The *CIsDomOfNom* relation is a restriction of *CIsDomOf* that ignores "accidental" relationships. Being a restriction of a Well Founded relation it is also a Well Founded relation. In any non-empty subset *X* of *CCarts* we can always choose a member of *X* whose domains are all basic domains or, at least, not members of *X*.

The next step in the proof of completeness is to restrict our attention to the more practical members of SetCartNom. Some members of SetCartNom have an infinite number of cartesian products, with infinite index sets. However, the various dialects of the NIAM notation require us to draw a separate symbol for each cartesian product and for each index. This implies that the number of cartesian products and indexes is necessarily finite. From now on we will restrict our attention to the subclass SetCartFin of SetCartNom defined as follows. The members of SetCartFin are the members of SetCartNom that have a finite number of cartesian products, each with a finite index set.

For each member of *C* of SetCartFin there is a construction sequence. That is, there is a sequence of members of SetCartFin with the following properties :

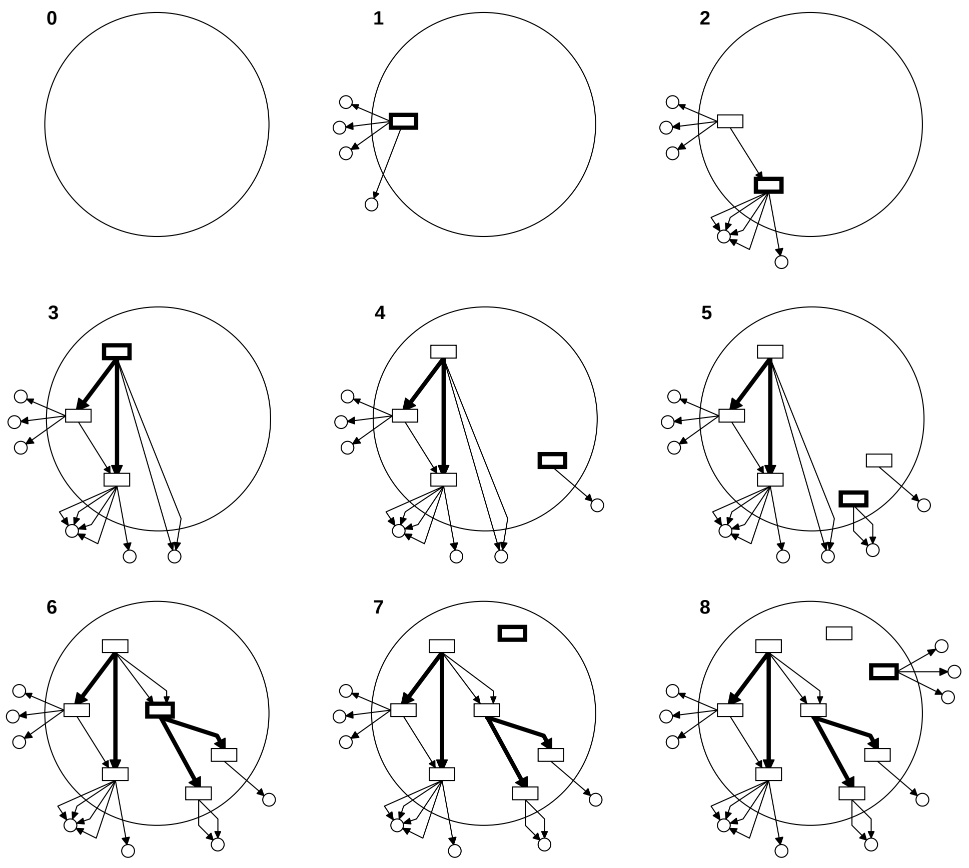
a) The first element of the sequence is the (unique) empty member of SetCartFin with no cartesian products;

b) The last element is *C*;

c) If *C'* is followed by *C''* in the sequence then *C''Carts* is obtained from *C'Carts* by adding one more cartesian product, say *c*, and *C''Noms* is obtained from *C'Noms* by adding the nominated indexes of *c*.

An example of a construction sequence is given in Figure 6.2.4 below.

Figure 6.2.4 A construction sequence



Notice that whenever a cartesian product with nominated indexes was added then its nominee domains were cartesian products that were added earlier in the sequence. This is essential if each element of the sequence is to be a member of SetCartFin, and hence a member of SetCartNom. (In any *C"* : SetCartNom every nominee domain must be a member of *C"Carts*). Thus at each point in the sequence the cartesian product to be added must be one whose domains are either basic domains or nominee domains that were added earlier. To put this the other way round, we have a set *X* of cartesian products that have not been added yet, and we must choose one whose nominee domains are not members of *X*. But, as we saw earlier, it is always possible to do this while any cartesian products remain to be added.

To sum up so far, for each *C* : SetCartFin the relation *CIsDomOfNom* is a Well Founded relation, and as a consequence there is a construction sequence for *C*. It appears that our original decision to describe data models as objects that can be built incrementally was a reasonable one.

The final step in the proof of completeness is to restrict our attention to the subclass SetCartFact of SetCartFin. We will define the subclass first, then justify the restrictions. The members of SetCartFact are the members of SetCartFin that obey the following four restrictions :

a) The nullary cartesian product Φ is excluded;

b) The basic domains are pairwise disjoint;

c) Each index belongs to the fixed set Roles;

d) Each member of a basic domain belongs to the fixed set Entities.

Restrictions (a) and (b) are NIAM restrictions. The nullary cartesian product Φ is not allowed. As we saw in Section 3.1.3 (Point 20), Φ is not a useful Fact Type. And it is a NIAM rule that the Entity Types used in a data model, which are modelled here by basic domains, are pairwise disjoint. It is also a NIAM rule that the index sets are pairwise disjoint, but this rule was incorporated in the definition of SetCartDis.

Restrictions (c) and (d) are modelling decisions. Remember that the purpose of SetCartFact is to model all the sets of cartesian products that can be specified by NIAM data models. The indexes and domain members used in SetCartFact are arbitrary modelling elements. The only requirement on these modelling elements is that we have enough of them for the purpose.

We know that each NIAM data model uses a finite number of indexes. Provided the set Roles is (countably) infinite then it has enough members to model the indexes of any NIAM data model.

On the other hand, we know that there are domains that are too large to be modelled by the members of any fixed set such as Entities. For instance, the power set of Entities is strictly larger than Entities. However, we also know that there are domains that cannot be what they appear to be. We saw in Section 3.2.3, Example 13, that the data model of a database holding data models has a domain apparently containing all possible domains. To avoid paradoxes that destroy the meaning of the data model we had to say that this domain actually contains a limited number of representatives. Each representative could be a description in words of a domain. There are only a countably infinite number of such descriptions.

Provided that the set Entities is large enough to model the real numbers then it is large enough to model most of the domains that arise in commerce and industry, and is large enough to model any domain of representatives. We will assume that any domain that appears to be too large to be modelled by Entities is, in fact, a domain of representatives. Given this assumption, then SetCartFact models all the sets of cartesian products that can be specified by NIAM data models. If the assumption seems unsatisfactory then note that a domain in a data model can be defined to be something that is too large to be modelled by any set. For instance, a database of Feature Notation models might have a domain declared to be "all" sets. Any set-theoretical model of data models will have some limitations. In SetCartFact they are made explicit rather than being implicit.

Now we can prove completeness. We wish to prove that any set of cartesian products that can be specified by a member of SetCartFact can also be specified by a member of DaMod0. The proof is given in full in the details section. We will give an outline here.

Suppose that we are given any member *C* of SetCartFact. We form a construction sequence for *C*. Thus we have a sequence *L* whose first element is the member of SetCartFact that has no cartesian products, and the last element is *C*. Each successive element is obtained by adding one more cartesian product and incrementing the set of nominated indexes as necessary. We then form a parallel sequence *P* of members of PreMod. There is an element of *P* for each element of *L*. We define *P* in such a way that each element of *P* is a member of DaMod0 that specifies a set of cartesian products that is the same as that of the corresponding element of *L*. Thus the last element of *P* is a member of DaMod0 that specifies the same cartesian products as *C* does, as desired.

The first element of the sequence *P* is the member of DaMod0 whose objects are *CDomsBasic*, with no roles associated with objects. Thus the first element introduces all the Entity Types, but no Fact Types. Each subsequent element of *P* introduces one additional set of roles and their role associations, with successive elements related by AddedRo. Of course, the additional set of roles is chosen to be the one that identifies the same cartesian product that was added to give the corresponding element of *L*.

Our definition of well-formed NIAM data models includes obviously unfinished data models that have Entity Types that are not a domain of any Fact Type. Clearly, DaMod0 models these data models as well.

To sum up, we have proved that DaMod0 models all well-formed NIAM conceptual data models, subject to two assumptions.

The first assumption is that cartesian products do not have circular definitions; no cartesian product is defined in terms of itself. This is a consequence of two sub-assumptions. The first is that tuples are packages that hold their values inside themselves. The second is that nothing can be inside itself.

The second assumption is that there is a limit on the size of Entity Types. What appear to be ultra-large Entity Types are assumed to be smaller, though possibly infinite, sets of representatives. Note that representatives also enable us to avoid paradoxes in some useful data models. They thereby become well-formed and so modelled by members of DaMod0.

We will finish with a brief discussion of the limitations of NIAM.

As we noted earlier, the common NIAM notations restrict each data model to a finite number of Fact Types with finite index sets. This is a clear-cut limitation, and one that will seldom preclude the use of NIAM.

A problem that has become apparent is more a warning that a limitation. NIAM allows an Entity Type to be anything that the data modeller wishes to talk about. Some care is needed when unusual Entity Types are defined. In particular, as we saw in Section 3.2.3, any kind of self-reference is dubious at best. E.g "All Entity Types"; "All database instances".

Another problem is that it is mathematically legitimate to assume that cartesian products can have circular definitions. It is not clear whether NIAM data models can provide a well defined specification of such objects. We will not attempt to analyse the consequences of allowing circular definitions.

6.3 Details

In this section we prove, given two reasonable assumptions, that DaMod0 is complete. That is, we prove that DaMod0 is a model of the core parts of all well-formed NIAM conceptual data models.

We start with the class of all sets of non-empty fact-style cartesian products. The CartProd operator, which gives us fact-style cartesian products, was defined in Section 4.3.2.

C : SetCart

Class of all sets of fact-style cartesian products

∗ *CCarts* : Set	A set of fact-style cartesian products

$CCond1$.: $\forall c : CCarts \bullet \exists F : \text{Set} \bullet \text{CartProd}(F) \neq \emptyset \wedge c = \text{CartProd}(F)$

As each cartesian product is non-empty we can retrieve its indexes, domains, and domain function by inspecting its tuples. Recall that for each T : Tuple the domain function *TDomf* is defined to be a set of couples. We can retrieve the domain function of a cartesian product by forming the union of its tuples' domain functions.

C : SetCart Secondary features : 1

Some features of each member of SetCart : Components

CInd	Index sets
CInd(*c*) : Set (*c* : *CCarts*)	Index set of the tuples of the cartesian product *c*

$CDefInd$.: $\forall c : CCarts \bullet CInd(c) =_d \bigcup \{ TI \mid T : c \}$

CDomf	Domain functions
$CDomf_c$: Set (*c* : *CCarts*)	Domain function of the tuples of the cartesian product *c*

$CDefDomf$.: $\forall c : CCarts \bullet CDomf_c =_d \bigcup \{ TDomf \mid T : c \}$

$CIndexes =_d \bigcup \{ CInd(c) \mid c : CCarts \}$	Set of all indexes occurring in *C*
$CDoms =_d \bigcup \{ \text{Ran}(CDomf_c) \mid c : CCarts \}$	Set of all domains occurring in *C*

There is an elementary classification of domains.

C : SetCart Secondary features : 2

Some features of each member of SetCart : An elementary classification of its domains

CDomsBase $=_\text{d}$ *CDoms* \ *CCarts*	Base domains : the domains that are not members of *CCarts*
CDomsCart $=_\text{d}$ *CDoms* $\cap$ *CCarts*	Domains that happen to be members of *CCarts*

The domain functions determine a dependency relation on the cartesian products.

C : SetCart Secondary features : 3

Some features of each member of SetCart : Structure

CIsDomOf : *CCarts* $\leftrightarrow$ *CCarts*	Given *x*, *y* : *CCarts* then *x CIsDomOf y* iff *x* is a domain of *y*

CDefIsDomOf .: $\forall x, y$: *CCarts* $\bullet$
x CIsDomOf y $\Leftrightarrow_\text{d}$ $\exists i$: *CInd*(*y*) $\bullet$ $x = CDomf_y(i)$

CPreds : *CCarts* $\rightarrow$ Pow(*CCarts*)	Given *c* : *CCarts* then *CPreds*(*c*) is the set of immediate predecessors of *c* w.r.t *CIsDomOf*

CDefPreds .: $\forall c$: *CCarts* $\bullet$
CPreds(*c*) $=_\text{d}$ { *x* : *CCarts* | *x CIsDomOf c* }

This relation is Well Founded, but the proof, given below, is dependent on some general assumptions about tuples and sets.

C : SetCart Properties : 3

Some properties of each member of SetCart : Well Founded

CProp3.1 .: *CIsDomOf* is a Well Founded relation
$\forall X \subseteq_\text{d}$ *CCarts* $\bullet$ $X \neq \emptyset \Rightarrow \exists c : X \bullet \forall x : X \bullet \neg\, x$ *CIsDomOf c*

There is a difficulty when proving *CProp3.1*. The proof makes use of a property of tuples that can vary with their representation. For objects defined in Feature Notation we do not specify how the various features are to be held together, hence the representation of tuples is left open to some extent. The proof of *CProp3.1* is split into two proof units. The first assumes a particular representation. The second contains an argument showing that *CProp3.1* is true for any reasonable representation of tuples that is general enough for our purposes.

We use an elementary property of Well Founded relations that will be used again later on. We give it the name WFProp1 and prove it separately. Recall that if R is a relation then RGr is its graph.

To prove
Property WFProp1 that any "sub-relation" of a Well Founded relation is also Well Founded. That is, to prove that

$\forall X, Y : \mathsf{Set} \bullet \forall R : X \leftrightarrow X \bullet \forall Q : Y \leftrightarrow Y \bullet$
$\quad [\, Y \subseteq X \;\wedge\; QGr \subseteq RGr \;\wedge\; R$ is Well Founded $] \;\Rightarrow\; Q$ is Well Founded.

The proof is almost immediate.

If for each non-empty subset $A \subseteq_{\mathrm{d}} Y \subseteq_{\mathrm{d}} X$ we have some $x : A$ such that

$\quad \forall y : A \bullet \neg\, y\, R\, x,$

then if $\forall x, y : Y \bullet y\, Q\, x \Rightarrow y\, R\, x$ we also have

$\quad \forall y : A \bullet \neg\, y\, Q\, x.$

We can conclude that WFProp1 is true.

□

To prove
Property *CProp3.1* that *CIsDomOf* is a Well Founded relation. That is, to prove that

$\forall C : \mathsf{SetCart} \bullet$
$\quad \forall X \subseteq_{\mathrm{d}} CCarts \bullet X \neq \emptyset \Rightarrow \exists c : X \bullet \forall x : X \bullet \neg\, x\; CIsDomOf\; c$

Assume that C : SetCart.

In outline, we will construct a Well Founded relation that is derived from the $\in$ relation, then show that the graph of *CIsDomOf* is a subset of the relation's graph.

Assume a specific representation for the members of Tuple. Recall that each T : Tuple has three features : an index set TI, a domain function $TDomf$, and a value function $TVal$. TI is defined to be any set; $TDomf$ and $TVal$ are defined to be sets of couples. Assume that features are held as tag, value couples, so that for any T : Tuple,

$\quad T =_{\mathrm{d}} \{ \langle \mathsf{I}, TI \rangle, \langle \mathsf{D}, TDomf \rangle, \langle \mathsf{V}, TVal \rangle \}$

where I, D, and V are three distinct constants. Assume also that couples are represented as Kuratowski couples, so for any sets x, y,

$\quad \langle x, y \rangle =_{\mathrm{d}} \{ \{x\}, \{x, y\} \}.$

If we have the fact-style cartesian products $c, c' : CCarts$ then $c'\; CIsDomOf\; c$ iff c' is a domain of c. That is to say, iff for any tuple $T : c$ there is some index $i : TI$ for which $TDomf(i) = c'$. If c' is a domain of c then c is a set of the form :

$c = \{ \ldots, T, \ldots \}$
$\quad = \{ \ldots, \{ \ldots, \langle \mathsf{D}, TDomf \rangle, \ldots \}, \ldots \}$
$\quad = \{ \ldots, \{ \ldots, \langle \mathsf{D}, \{ \ldots, \langle i, c' \rangle, \ldots \} \rangle, \ldots \}, \ldots \}.$

We then have the chain of memberships

$c' \in \{ i, c' \} \in \langle i, c' \rangle \in TDomf \in \{ \mathsf{D}, TDomf \} \in \langle \mathsf{D}, TDomf \rangle \in T \in c.$

If we vary *c* and *c'* then whenever *c' CIsDomOf c* is true a membership chain of this kind will exist. We wish to construct a relation $\in_K^+$ for which $c' \in_K^+ c$ whenever there is a membership chain from *c'* to *c*, and hence whenever *c' CIsDomOf c*.

First, we construct a set *K* containing all the sets that could occur in such membership chains, and many more sets besides. Define the term $\mathsf{U}^n X$ by recursion on Nat as

$\mathsf{U}^0 X =_{\mathrm{d}} X;\ \mathsf{U}^{n+1} X =_{\mathrm{d}} \bigcup \mathsf{U}^n X; \qquad (n : \mathsf{Nat}, X : \mathsf{Set}).$

Now define the set *K* as

$K =_{\mathrm{d}} \bigcup \{ \mathsf{U}^n\ CCarts \mid n : \mathsf{Nat} \}.$

Thus *K* is $CCarts \cup \bigcup CCarts \cup \bigcup\bigcup CCarts \cup \bigcup\bigcup\bigcup CCarts \ \ldots$;
i.e the cartesian products, the tuples, the members of tuples, … .

Next, we define the relation $\in_K : K \leftrightarrow K$ as the restriction of $\in$ to *K*, thus

$\forall x, y : K \bullet x \in_K y \Leftrightarrow_{\mathrm{d}} x \in y.$

Form the transitive closure of $\in_K$ to give the relation $\in_K^+ : K \leftrightarrow K$. This is the relation with the property

$\forall c', c : CCarts \bullet c'\ CIsDomOf\ c \Rightarrow c' \in_K^+ c.$

Enderton gives a proof that $\in_K$ is Well Founded (for any set *K*), and a proof that the transitive closure of a Well Founded relation is also Well Founded (Enderton [1977], p242-243).

We now apply WFProp1. We have $\in_K^+ \in K \leftrightarrow K$,
$CIsDomOf \in CCarts \leftrightarrow CCarts$, $CCarts \subseteq K$, $CIsDomOfGr \subseteq \mathsf{Gr}(\in_K^+)$, and $\in_K^+$ is Well Founded; therefore *CIsDomOf* is Well Founded. Hence we can conclude for all *C* : SetCart that *CProp3.1* is true.

□

Note that the proof that $\in_K$ is Well Founded makes essential use of the Axiom of Foundation. In a set theory that lacks this axiom the property *CProp3.1,* and the completeness property *CPropFact1.3* which relies on *CProp3.1,* are not necessarily true.

Note also that this proof has used a specific representation of features, tuples, and couples.

To "prove"
by a plausibility argument that *CProp3.1* is true whatever representation of tuples might have been used.

The proof of *CProp3.1* assumed that a very specific representation of features, tuples, and couples is being used. Is the property true for other representations? We will give two counter-examples of representations where the property is not true, but we will argue that these representations are not appropriate here. We will then show that the property is true if we make a reasonable assumption about suitable representations of tuples.

The first example uses the standard representation of 1-tuples. For any set x define the 1-tuple $\langle x\rangle$ whose single element is x by $\langle x\rangle =_d x$. The unary cartesian product whose one domain is X is then $\{ \langle x\rangle \mid x : X \} = X$. The cartesian product is a domain of itself, and the "is a domain of" relation is not Well Founded. However, in data modelling we wish to distinguish the cartesian product $\{ \langle x\rangle \mid x : X \}$ from the cartesian product $\{ \langle\langle x\rangle\rangle \mid x : X \}$. We would have to use tags, alias indexes, to distinguish between these cartesian products or between their tuples. We do the latter in Tuple.

For the second example, observe that the only requirement for tuples is that there is a constructor function for assembling tuples from values and a projection function to regain the values. We could define these functions in unusual ways. For instance, suppose we have the set $\mathsf{C} =_d \{ \{\mathsf{T1}, \mathsf{T2}\}, \{\mathsf{T3}, \mathsf{T4}\} \}$ of cartesian products. We can define the projection function π for one co-ordinate to be $\pi(\mathsf{T1}) =_d \mathsf{T3}$, $\pi(\mathsf{T2}) =_d \mathsf{T4}$, and $\pi(\mathsf{T3}) =_d \mathsf{T1}$, $\pi(\mathsf{T4}) =_d \mathsf{T2}$, with the result that the cartesian product $\{\mathsf{T1}, \mathsf{T2}\}$ is a domain of the cartesian product $\{\mathsf{T3}, \mathsf{T4}\}$ and vice versa. Again the "is a domain of" relation is not Well Founded. Notice that we restricted ourselves to a finite set of tuples and so were able to define π in any way we wished. More generally, if we restrict ourselves to a specific set of tuples then we are free to define π in unusual ways.

However, in our investigation into conceptual data models we wish to consider tuples constructed from values ranging over all sets. Even though we started with the fixed sets Entities and Roles we are not interested in any properties that depend on these sets having a particular membership. As a consequence, any projection function π that we use must be defined by some formula α, and α must not select any tuples as special cases. π is defined by the sentence

$$\forall T \bullet \forall v \bullet \pi(T) = v \Leftrightarrow_d \alpha.$$

There are restrictions on α : see Enderton [1972], p154-156 on non-creative extensions to a theory. In outline, α must describe a rule for using set operations to obtain a unique value v given any tuple T. As the rule has only the membership of T to work on it seems reasonable to restrict ourselves to rules that require v to be a member of T, or a member of a member, or a member of a member of a member, … . With this general restriction we can prove that the "is a domain of" relation is Well Founded, as follows.

Assume that whenever any set v is an element of a tuple T then there is a membership chain $v \in \ldots \in T$.

For any set A of tuples define the relation *IsElOf* on A where T' *IsElOf* T iff T' is an element of T. By the assumption, whenever T' *IsElOf* T there is a membership chain $T' \in \ldots \in T$. We can form the set $(A \cup \bigcup A \cup \bigcup\bigcup A \cup \bigcup\bigcup\bigcup A \cup \ldots)$ and the relation derived from $\in$ as in the proof of *CProp3.1* to prove that *IsElOf* is Well Founded. Consequently, if $A \neq \emptyset$ then there is some tuple of A none of whose elements is a member of A. Thus for any non-empty set X of non-empty cartesian products there is some $c : X$ one of whose tuples has no elements belonging to the set $\bigcup X$ of tuples. Hence none of c's domains can be a member of X.

We can conclude for any set C of cartesian products that, given the reasonable assumption, the "is a domain of" relation on C is Well Founded.

□

Remember, though, that this second proof unit still makes essential use of the Axiom of Foundation.

Next, we specialise SetCart to the cases where the cartesian products have pairwise disjoint index sets.

C : SetCartDis (Specialised SetCart)

Subclass of SetCart where index sets are distinct and disjoint

$*$ *C*[SetCart] : SetCart	Set of cartesian products (with features *CCarts*, *CCond1*, etc)

$CCondDis1$.: $\forall c, c' : CCarts \bullet c \neq c' \Rightarrow CInd(c) \cap CInd(c') = \varnothing$

The relation from indexes to domains is now a function.

C : SetCartDis Secondary features : 1

A feature of each member of SetCartDis : Overall domain function

$CInDom : CIndexes \to CDoms$	Given $i : CIndexes$ then $CInDom(i)$ is the unique domain associated with i

$CDefInDom$.:
$\forall c : CCarts \bullet \forall i : CInd(c) \bullet CInDom(i) =_{\mathrm{d}} CDomf_c(i)$

SetCartDis has two immediate properties.

C : SetCartDis Properties : 1

Some properties of each member of SetCartDis

$CPropDis1.1$.: Index sets identify cartesian products
$\forall c, c' : CCarts \bullet c = c' \Leftrightarrow CInd(c) = CInd(c')$

$CPropDis1.2$.: *CInDom* is well defined as a total function

---

Next, we extend SetCartDis so that we can highlight some of the indexes. Recall from Section 5.4.2 that [[ ]] is the image operator.

C : SetCartNom

Class SetCartDis extended to include a set of nominated indexes

∗ *C*[SetCartDis] : SetCartDis	Set of cartesian products (with features *CCarts*, *CCond1*, *CCondDis1*, etc)
∗ *CNoms* $\subseteq_d$ *CIndexes*	Nominated indexes

CCondNom1 .: *CInDom*[[*CNoms*]] $\subseteq$ *CCarts*

We now have a further classification of domains.

C : SetCartNom Secondary features : 1

Some features of each member of SetCartNom : Further classification of domains

CDomsNom $=_d$ *CInDom*[[*CNoms*]]	Nominee domains
CDomsBasic $=_d$ *CInDom*[[*CIndexes* - *CNoms*]]	Basic domains
CDomsAcc $=_d$ *CDomsBasic* - *CDomsBase*	"Accidental" domains

---

Nomination leads to a restricted dependency relation on the cartesian products.

C : SetCartNom Secondary features : 2

Some features of each member of SetCartNom : Structure, taking nomination into account

CIsDomOfNom : *CCarts* $\leftrightarrow$ *CCarts*	Given *x*, *y* : *CCarts* then *x CIsDomOfNom y* iff *x* is a nominee domain of *y*

CDefIsDomOfNom .: $\forall x, y$: *CCarts* $\bullet$
x CIsDomOfNom y $\Leftrightarrow_d$ $\exists i$: *CInd*(*y*) $\bullet$ $x = CDomf_y(i) \wedge i \in CNoms$

CPredsNom : *CCarts* $\rightarrow$ Pow(*CCarts*)	Given *c* : *CCarts* then *CPredsNom*(*c*) is the set of immediate predecessors of *c* w.r.t *CIsDomOfNom*

CDefPredsNom .: $\forall c$: *CCarts* $\bullet$
CPredsNom(*c*) $=_d$ { *x* : *CCarts* | *x CIsDomOfNom c* }

Given the assumptions that ensure that *CIsDomOf* is Well Founded then it is immediate from WFProp1 that *CIsDomOfNom* is also Well Founded.

C : SetCartNom   Properties : 2

Some properties of each member of SetCartNom : Well Founded w.r.t *CIsDomOfNom*

CPropNom2.1 .: $CIsDomOfNomGr \subseteq CIsDomOfGr$

CPropNom2.3 .: *CIsDomOfNom* is a Well Founded relation
$\forall X \subseteq_d CCarts \bullet X \neq \varnothing \Rightarrow \exists c : X \bullet \forall x : X \bullet \neg\, x\ CIsDomOfNom\ c$

Next, we specialise SetCartNom to the cases where we have a finite set of cartesian products with finite arities.

C : SetCartFin   (Specialised SetCartNom)

Subclass of SetCartNom where *CCarts* is a finite set of cartesian products whose index sets are finite. **Note** Domains and indexes can still be any sets.

∗ *C*[SetCartNom] : SetCartNom	Set of cartesian products and nominations (with features *CCarts*, *CNoms*, *CCond1*, *CCondDis1*, *CCondNom1*, etc)

CCondFin1 .: IsFinite(*CCarts*)

CCondFin2 .: $\forall c : CCarts \bullet$ IsFinite(*CInd*(*c*))

It is immediate that several secondary features are also finite.

C : SetCartFin   Properties : 1

Some properties of each member of SetCartFin : Finiteness

CPropFin1.1 .: IsFinite(*CIndexes*) $\wedge$ IsFinite(*CDoms*) $\wedge$ IsFinite(*CDomsBasic*)

We wish to prove that there is a construction sequence for every member of SetCartFin. We do this by defining an operator that returns a construction sequence given any member of SetCartFin, then proving that the operator is well defined. The operator must use a choice function to choose cartesian products one by one. We start by defining all the possible choice functions.

C : SetCartFin Secondary features : 1

A feature of each member of SetCartFin : Choice functions

$CChos \subseteq_d \mathrm{Pow}(CCarts) \to CCarts$ Set of all possible choice functions for the set *CCarts*

$CDefChos$.: $CChos =_d \{ H : \mathrm{Pow}(CCarts) \to CCarts \mid \forall X \subseteq_d CCarts \bullet X \neq \varnothing \Rightarrow H(X) \in X \}$

There is no total function from a non-empty set to an empty set. Consequently, if *CCarts* is empty then *CChos* is empty. However, if *CCarts* is not empty then *CCarts* is finite so choice functions for *CCarts* exist and *CChos* is not empty.

C : SetCartFin Properties : 2

A property of each member of SetCartFin : Choice functions

$CPropFin2.1$.: $CChos = \varnothing \Leftrightarrow CCarts = \varnothing$

A construction sequence is defined here to be a member of Scheurer's class ListV of valued lists. Recall from Section 4.5.4 that the primary variable features of any list *L* : ListV are a set *LPts* of points, a set *LVals* of values, a value function *LV*, and a successor function *LS*. The first and last points are *LFst* and *LLst*, when they exist.

All the values occurring in the list must be members of *LVals*, which is a little awkward here. We would like to say that *LVals* is SetCartFin but SetCartFin is a proper class. We will see that there is a set that will do instead.

The operator takes two arguments. One is any *C* : SetCartFin; the other is a choice function for *C*, provided *C* has a choice function. If *CCarts* = ∅, and so has no choice function, we use the dummy argument ∅.

We now define the operator and then prove that it is well defined. Recall that the "+" symbol in place of "∪" signals the union of disjoint sets.

ConsSeqFin

Class function which when given any C : SetCartFin and H : *CChos* (or a dummy value) returns a construction sequence that ends at C

Given any C : SetCartFin, $H : (CChos \cup \{\varnothing\})$ with

(Pre 1) $H = \varnothing \Leftrightarrow CCarts = \varnothing$

then

ConsSeqFin(C, H) $=_d L$ where L : ListV and

$LVals =_d \{ C' : \text{SetCartFin} \mid C'Carts \subseteq CCarts \} \wedge$ (a)

$LV(LFst)Carts =_d \varnothing \wedge$ (b)

$LV(LLst) =_d C \wedge$ (c)

$[H = \varnothing \Rightarrow LSDef = \varnothing] \wedge$ (d)

$[\forall i : LSDef \bullet$
Let $C' =_d LV(i)$, $C'' =_d LV(LS(i))$
Let $c =_d H(\{ x : (CCarts - C'Carts) \mid CPredsNom(x) \subseteq C'Carts \})$

$C''Carts =_d C'Carts + \{c\} \wedge$ (e)
$C''Noms =_d CNoms \cap C''Indexes$ (f)
$]$

To Prove
That ConsSeqFin is well-defined. That is, for any C and H satisfying the preconditions there exists a list L : ListV obeying the definition of ConsSeqFin(C, H), and L is unique up to isomorphism.

The proof is much like the proof that CompSeq is well defined (Section 4.5.4).

Assume that C : SetCartFin, $H : (CChos \cup \{\varnothing\})$, and that (Pre 1) is true.

First, isomorphism. Assume that a list L : ListV obeying the definition of ConsSeqFin(C, H) exists. The proof is by induction on *LPts*. We wish to prove that the first value of L and all successive values are uniquely determined independently of *LPts*. Recall that any C' : SetCartFin is uniquely determined when *C'Carts* and *C'Noms* are given.

By item (b) in the definition of ConsSeqFin the first value in the list is some C' : SetCartFin for which $C'Carts = \varnothing$. But then $C'Indexes = \varnothing$. As $C'Noms \subseteq C'Indexes$ then $C'Noms = \varnothing$. Thus the first value, C', is uniquely determined.

Now let C', C'' be successive values as in items (e) and (f) of the definition. Assume that C' is uniquely determined. Then by (e) *C''Carts* is uniquely determined and hence so is *C''Indexes*. Therefore by (f) *C''Noms* is also uniquely determined and so C'' is uniquely determined. This value C'' is independent of the choice of points.

By (c) we have that the list ends at an occurrence of the value *C*. We wish to prove that it must end at the first occurrence. If *CCarts* = ∅ then *H* = ∅ by (Pre 1) and by (d) the list has only one element, which is *C*; the list ends at the first occurrence of *C*.

Suppose now that *CCarts* ≠ ∅ so *H* ∈ *CChos* and suppose that the list does not end at the first occurrence of the value *C*. Then there is a point *i* : *LSDef* for which *LV*(*i*) = *C*. Let *C''* be the next value, *LV*(*LS*(*i*)). By (e) we have *C''Carts* = *CCarts* + {*c*} where *c* = *H*(∅). By the definition of *H* as a member of *CChos* we have *c* ∈ *CCarts*. But the use of the "+" symbol in (e) signals the union of disjoint sets, so we have *c* ∉ *CCarts*. We conclude that *i* ∉ *LSDef*, so the list must end here at the first occurrence of *C*. This conclusion is independent of the choice of points.

We can conclude that any lists obeying the definition of ConsSeqFin(*C*, *H*) are isomorphic.

Second, existence. It suffices to prove that the definition has the following properties :

E1) The definition assigns a member of SetCartFin to each point, a member that obeys the requirements for LVals.

E2) The value *C* is assigned at least once.

E1 : We wish to prove that the definition assigns a member of SetCartFin to each point.

Clearly there is a *C'* : SetCartFin such that *C'Carts* = ∅. Thus item (b) of the definition assigns a member of SetCartFin to the first point, with *C'Carts* ⊆ *CCarts*.

Now assume that we are given that the point *i* has a successor point *i'*, that *i* is assigned *C'* : SetCartFin, and that *C'Carts* ⊆ *CCarts*. We wish to prove that there is some *C''* : SetCartFin that obeys definition items (e) and (f). Clearly *C''Carts* is a subset of *CCarts* and so obeys all the required conditions from *C''Cond1* through to *C''CondFin2*.

For *C''Noms* we require that *C''Noms* ⊆ *C''Indexes*; and it is by item (f) of the definition. Recall that the function *C''InDom* gives the domain of each index belonging to *C''Indexes*. For *C''Noms* we also require by *C''CondNom1* that *C''InDom*[[*C''Noms*]] ⊆ *C''Carts*.

For some *c* : *CCarts* we have *C''Carts* = *C'Carts* + {*c*}. As the index sets of *C* are distinct and disjoint we have *C''Indexes* = *C'Indexes* + *CInd*(*c*). By assumption *C'* ∈ SetCartFin so we have *C'InDom*[[*C'Noms*]] ⊆ *C'Carts*. For any index *j* : *C''Noms* either *j* ∈ *C'Noms* or *j* ∈ (*CInd*(*c*) ∩ *CNoms*). If the former, then *C''InDom*(*j*) = *C'InDom*(*j*) ∈ *C'Carts*. If the latter, by the choice of *c* in item (e) we have *CPredsNom*(*c*) ⊆ *C'Carts*, which is to say that the domain of each nominated index, if any, of *c* is a member of *C'Carts*. Therefore, *C''InDom*(*j*) ∈*C'Carts*. Thus we have *C''InDom*[[*C''Noms*]] ⊆ *C'Carts* ⊆ *C''Carts*. *C''Noms* obeys *C''CondNom1* as desired.

Thus *i'* is assigned a member of SetCartFin; also *C''Carts* ⊆ *CCarts*.

We can conclude that the definition assigns a member of SetCartFin to each point, one that is a member of *LVals*.

E2 : We wish to prove that the value *C* occurs at least once.

If $CCarts = \varnothing$ then the list starts with an occurrence of *C*.

Now assume that $CCarts \neq \varnothing$ and hence that $H \neq \varnothing$. Observe that in item (e) of the definition of ConsSeqFin we have a choice function *H* acting on a subset of *CCarts*. As *CIsDomOfNom* is a Well Founded relation we can be sure that for the *C'* : ConsSeqFin defined in item (e) there is an *x* : (*CCarts* - *C'Carts*) such that $CPredsNom(x) \subseteq C'Carts$ whenever $C'Carts \subset CCarts$. Thus *H* chooses each member of *CCarts* in turn exactly once and there is no reason to end the list until *C* is reached. As *CCarts* is finite *C* will be reached in a finite number of steps.

We conclude that the value *C* occurs at least once.

Finally, from E1 and E2 we can conclude that there is a list *L* : *ListV* that obeys the definition of ConsSeqFin(*C*, *H*). Altogether, we can conclude that ConsSeqFin is well defined (up to isomorphism).

□

Note that the proof, part E2, assumes that *CIsDomOfNom* is a Well Founded relation for every *C* : SetCartFin. The proof makes essential use of the assumptions needed to prove property *CProp3.1*.

Next we specialise SetCartFin to the cases where indexes are drawn from the fixed set Roles and basic domain members from the fixed set Entities. We also include some NIAM rules.

C : SetCartFact (Specialised SetCartFin)

Subclass of SetCartFin where index sets are restricted to being non-empty subsets of Roles, and basic domains are restricted to being pairwise disjoint subsets of Entities.

∗ *C*[SetCartFin] : SetCartFin	Set of cartesian products and nominations (with features *CCarts*, *CNoms*, *CCond1*, *CCondDis1*, *CCondNom1*, *CCondFin1*, *CCondFin2*, etc)

CCondFact1 .: $\forall c : CCarts \bullet CInd(c) \subseteq \text{Roles} \wedge CInd(c) \neq \varnothing$

CCondFact2 .: $\forall E : CDomsBasic \bullet E \subseteq \text{Entities}$

CCondFact3 .: IsPairwiseDisjoint(*CDomsBasic*)

Finally, we state and prove that DaMod0 can do anything that SetCartFact can do.

Recall from Sections 4.2.3 and 4.3.2 that DaMod0 is a subset of PreMod, and that each *P* : PreMod has the two primary variable features *PObjs* and *PConn*. Also, *PRoSets* is the set of those members of *PObjs* that are index sets, alias subsets of Roles. For each *D* : DaMod0 and *t* : *DRoSets*, *DCart*(*t*) is the cartesian product identified by *t*.

C : SetCartFact Properties : 1

Some properties of each member of SetCartFact

CPropFact1.3 .: DaMod0 is complete
$\exists D$: DaMod0 $\bullet$ { *DCart*(*t*) | *t* : *DRoSets* } = *CCarts*

To prove
Property *CPropFact1.3* that DaMod0 is complete. That is, to prove that
$\forall C$: SetCartFact $\bullet$ $\exists D$: DaMod0 $\bullet$ { *DCart*(*t*) | *t* : *DRoSets* } = *CCarts*

Assume that *C* : SetCartFact.

We will construct a member of DaMod0 with the stated property. In outline, we first form a list *L* : ListV that is a construction sequence for *C*. From *L* we derive a list *P* : ListV of members of PreMod. The two lists have the same length. We then prove that *P* is a completion sequence for some member of DaMod0 and that this member has the stated property.

Assume that $H =_d \emptyset$ if $CCarts = \emptyset$ and that *H* : *CChos* otherwise. Define the list *L* : ListV as the construction sequence $L =_d$ ConsSeqFin(*C*, *H*). Recall that *L* is a list of members of SetCartFin whose last element is *C*.

We wish to form a list whose elements are defined by recursion on *LPts*, and then prove their properties by induction on *LPts*. To make the proof easier to read we will give the recursion steps and induction steps together.

Define the list *P* : ListV of members of PreMod as follows :
$PPts =_d LPts$;
$PVals =_d$ PreMod;
$PS =_d LS$.
PV is derived from *LV*; it is defined below by recursion on *LPts* and its properties are proved by induction on *LPts*.

Define the formulas α and α' to be
$\alpha =_{SYM} PV(i) \in \text{DaMod0} \wedge \{ PV(i)Cart(t) \mid t : PV(i)RoSets \} = LV(i)Carts$;
$\alpha' =_{SYM} \alpha^{i}_{LS(i)}$, meaning that *LS*(*i*) is substituted for *i* in α.

Assume that *i* : *LPts*. There are two cases to consider.

C1 Case *i* = *LFst* = *PFst*
Define the first element of *P* by :

$PV(i)Objs =_d CDomsBasic$;
$PV(i)ConnDef =_d \emptyset$.

First, we must confirm that there is such a member of PreMod. By *CCondFact2* each member of *CDomsBasic* is a subset of Entities, so

CDomsBasic is a subset of Objects, as required. There is exactly one partial function from Roles to Objects whose definition domain is empty. We conclude that the two primary variable features of *PV*(*i*) are well defined.

Next, we wish to prove that *PV*(*i*) is a member of DaMod0. This will be so if we can form a construction sequence starting at EmpDaMod0, adding one member of *CDomsBasic* at a time, and ending at *PV*(*i*), with successive elements of the sequence related by AddedEn. But this is possible : by *CCondFact2* each member of *CDomsBasic* is a subset of Entities; as each is a domain of a non-empty cartesian product it is not empty; by *CCondFact3*, *CDomsBasic* is pairwise disjoint; and by *CPropFin1.1*, *CDomsBasic* is finite so a finite construction sequence can be defined. We conclude that $PV(i) \in$ DaMod0.

Finally, we wish to prove that *PV*(*i*) and *LV*(*i*) define the same set of cartesian products. By the definition of ConsSeqFin we have $LV(i)Carts = \varnothing$. The sets Entities and Roles are disjoint so there is no member of *PV*(*i*)*Objs* that is a subset of Roles, hence $PV(i)RoSets = \varnothing$ and $\{ PV(i)Cart(t) \mid t : PV(i)RoSets \} = \varnothing$ also. We conclude that they define the same, empty, set of cartesian products.

Altogether, we conclude that $i = LFst \Rightarrow \alpha$.

C2 Case $i \in LSDef$

Then *i* has a successor $i' =_d LS(i)$ and $H \neq \varnothing$. We are given *PV*(*i*) and we are to define *PV*(*i'*). From the definition of ConsSeqFin we see that $LV(i')Carts = LV(i)Carts + \{c\}$ where $c =_d H(\{ \dots \}) \in CCarts$. Define the next element of *P* by :

$PV(i')Objs =_d PV(i)Objs \cup \{CInd(c)\}$;
$PV(i')ConnDef =_d PV(i)ConnDef \cup CInd(c)$;

$\forall r : PV(i')ConnDef \bullet$

$PV(i')Conn(r) =_d PV(i)Conn(r)$	if $r \notin CInd(c)$;
$PV(i')Conn(r) =_d CInDom(r)$	if $r \in CInd(c) \wedge r \notin CNoms$;
$PV(i')Conn(r) =_d CInd(CInDom(r))$	if $r \in CInd(c) \wedge r \in CNoms$.

Now assume that α is true.

First, we must again confirm that there is such a member of PreMod. By α, $PV(i) \in$ DaMod0 and, by *CCondFact1*, *CInd*(*c*) is a subset of Roles so *PV*(*i'*)*Objs* is a subset of Objects, as required. Also we have that *PV*(*i'*)*ConnDef* is a subset of Roles, as required. *PV*(*i'*)*Conn* cannot fail to be functional as it is defined via a partition of *PV*(*i'*)*ConnDef*.

We must now confirm that the range of *PV*(*i'*)*Conn* is defined to be a subset of Objects. Assume that $r : PV(i')ConnDef$.
If $r \notin CInd(c)$ then $PV(i)Conn(r) \in$ Objects as $PV(i) \in$ DaMod0.
If $r \in CInd(c) \wedge r \notin CNoms$ then $CInDom(r) \in CDomsBasic$ and so is a member of Objects, as noted in case C1.
If $r \in CInd(c) \wedge r \in CNoms$ then $CInDom(r) \in CCarts$ by *CCondNom1* so *CInd*(*CInDom*(*r*)) is defined, and that in turn is a subset of Roles by

CCondFact1. We conclude that the two primary variable features of *PV*(*i*′) are well defined.

Next, we wish to prove that *PV*(*i*) AddedRo *PV*(*i*′) and hence that *PV*(*i*′) $\in$ DaMod0. *PV*(*i*′)*Objs* was obtained from *PV*(*i*)*Objs* by adding the set *CInd*(*c*). By *CCondFact1*, *CInd*(*c*) is a non-empty subset of Roles. By *CCondFin2* it is finite. From the definition of ConsSeqFin we see that *c* has not been introduced into *LV*(*i*)*Carts* prior to the point *i*′. Hence, by *CCondDis1*, *CInd*(*c*) is distinct and disjoint from any index set belonging to *PV*(*i*)*Objs*; as Roles and Entities are disjoint it is therefore distinct and disjoint from all members of *PV*(*i*)*Objs*.

By the definition of *PV*(*i*′), *PV*(*i*′)*Conn* is unchanged from *PV*(*i*)*Conn* for roles not belonging to *CInd*(*c*). For each role *r* : *CInd*(*c*) we wish to prove that *PV*(*i*′)*Conn*(*r*) $\in$ *PV*(*i*)*Objs*. If *r* $\notin$ *CNoms* then *CInDom*(*r*) is a basic domain and so was introduced into *PV*(*i*)*Objs* at *PFst*. If *r* $\in$ *CNoms* then we wish to prove that *CInd*(*CInDom*(*r*)) is a member of *PV*(*i*)*Objs*. But in the proof that ConsSeqFin is well defined, under E1, we proved that *LV*(*i*′)*InDom*[[*LV*(*i*′)*Noms*]] $\subseteq$ *LV*(*i*)*Carts*. *r* $\in$ *LV*(*i*′)*Noms* so *LV*(*i*′)*InDom*(*r*) = *CInDom*(*r*) is a cartesian product encountered earlier in the sequence. Thus its index set *CInd*(*CInDom*(*r*)) is indeed a member of *PV*(*i*)*Objs*. We now have all the conditions required for *PV*(*i*) AddedRo *PV*(*i*′). We conclude that *PV*(*i*′) $\in$ DaMod0.

Finally, we wish to prove that *PV*(*i*′) and *LV*(*i*′) define the same set of cartesian products. By α, *PV*(*i*) and *LV*(*i*) define the same set of cartesian products. *LV*(*i*′) defines one additional cartesian product, namely *c*. By the preservation of cartesian products, property *PV*(*i*)*Prop4.14* in Section 4.4.2, *PV*(*i*′) defines the cartesian products of *LV*(*i*) and one additional cartesian product, $c' =_{\mathrm{d}}$ *PV*(*i*′)*Cart*(*CInd*(*c*)). We wish to prove that $c = c'$. It suffices to prove that they have the same index sets and that each index is associated with the same domain.

The index set of *c* is *CInd*(*c*), and so is the index set of *c*′. For each index *r* : *CInd*(*c*) the domain assigned to *r* in the tuples of *c* is *CInDom*(*r*). There are two cases to consider : when *r* is a member of *CNoms* and when it is not.

If *r* $\notin$ *CNoms* then the domain assigned to *r* is a member of *CDomsBasic* so it is a subset of Entities. From the definition of *PV*(*i*′) we have *PV*(*i*′)*Conn*(*r*) = *CInDom*(*r*). From the definition of *PV*(*i*′)*Cart* in Section 4.3.2, as *PV*(*i*′)*Conn*(*r*) $\subseteq$ Entities then for the tuples of *c*′ the domain assigned to *r* is *PV*(*i*′)*Conn*(*r*), which is *CInDom*(*r*). The two domains are the same.

If *r* $\in$ *CNoms* then we have *PV*(*i*′)*Conn*(*r*) = *CInd*(*CInDom*(*r*)), which is a subset of Roles and a member of *PV*(*i*)*Objs*. From the definition of *PV*(*i*′)*Cart* in Section 4.3.2, as *PV*(*i*′)*Conn*(*r*) $\subseteq$ Roles then for the tuples of *c*′ the domain assigned to *r* is *PV*(*i*′)*Cart*(*PV*(*i*′)*Conn*(*r*)). Now this is a cartesian product defined by *PV*(*i*) so by α it is member of *LV*(*i*)*Carts*. By *CCondDis1* this cartesian product is uniquely determined by its index set, which is *CInd*(*CInDom*(*r*)). But for the tuples of *c* the domain assigned to *r* is the cartesian product *CInDom*(*r*), which is uniquely determined by its index set, *CInd*(*CInDom*(*r*)). The two domains are the same.

We conclude that $c = c'$, and hence that $PV(i')$ and $LV(i')$ define the same set of cartesian products.

Thus, combining these results we can conclude that $\forall i : LSDef \bullet \alpha \Rightarrow \alpha'$.

From case C1 and C2 we conclude that α is true for all $i : LPts$, and hence we can conclude for all C : SetCartFact that property *CPropFact1.3* is true. In other words, DaMod0 does everything that SetCartFact can do.

□

Note that the proof assumes that there is a construction sequence for every C : SetCartFact. This in turn assumes that *CIsDomOfNom* is Well Founded. The proof makes essential use of the assumptions needed to prove property *CProp3.1*.

7 Core model extended

So far, we have concentrated on modelling the core part of NIAM conceptual data models, the part that defines a set of cartesian products. Now we will extend the model to include the other information, such as names and constraint symbols, that can appear in a data model. This is mostly straightforward, with few restrictions on editing operations.

We start with the simplest topics : the names of roles, Entity Types, and Fact Types, and the marks that indicate Label Types and derived Fact Types (Section 7.1). In order to define database constraints we must define database instances and some derived values called populations (Section 7.2). We then model some of the constraint symbols that NIAM uses (Section 7.3). As different dialects of the NIAM notation use different constraint symbols we will not attempt to cover all of them. We will only do sufficient to illustrate the principles. Finally, we show, very briefly, how the model can be used to describe database operations (Section 7.4).

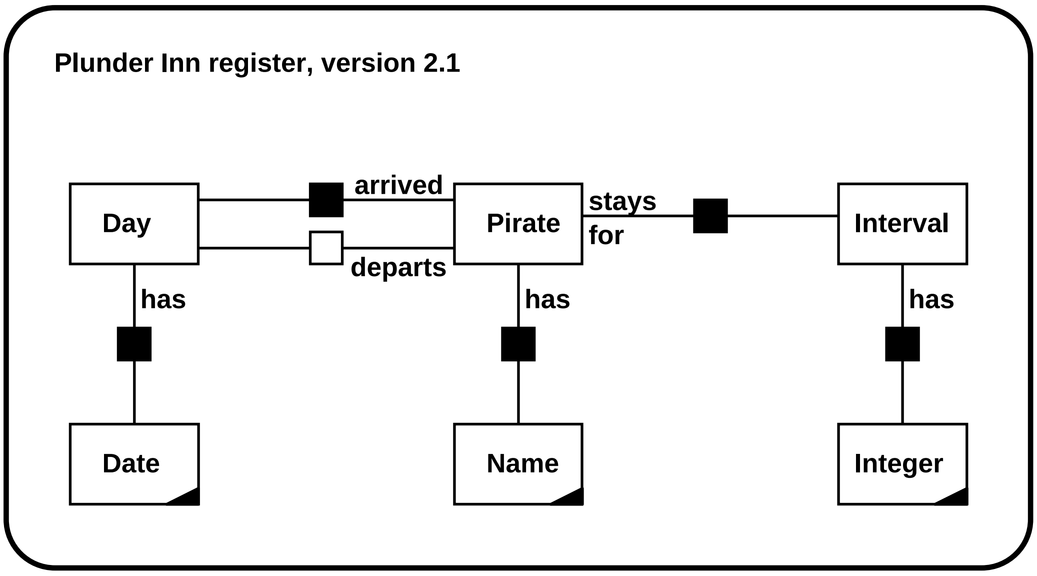
All these extensions are collected together in the Feature Notation definition of the set DaMod1 (Section 7.5). Each member of DaMod1 consists of a member of DaMod0, defining a core model, and the additional information outlined above. The definition includes ellipsis symbols indicating where a practical extended model would, no doubt, require additional features.

Although editing operations are discussed they are not defined in detail.

7.1 Names and marks

Figure 7.1.1 shows a Well Formed data model that we have seen before. We know that the core part of the data model is modelled by some member of DaMod0. However, the data model contains extra information that we have not modelled so far. Some objects and roles have been given visible names, some small squares have white centres indicating that they identify derived Fact Types, some large rectangles have black triangles indicating that they are Label Types, and the data model has a title and a version number.

Figure 7.1.1 An incomplete NIAM data model (Drawn in the "UMIST" dialect of the notation)



Recall from Section 4.2 that each *D* : DaMod0 has the two primary variable features

DObjs	The set of objects; each is either a subset of Entities, modelling an Entity Type, or a subset of Roles, modelling the index set of a Fact Type;
DConn	The function that associates roles with objects.

Recall from Section 4.3 that *D* has many secondary variable features, including

DEnSets	The members of *DObjs* that are subsets of Entities, i.e the objects occurring in *D* that model Entity Types;
DRoSets	The members of *DObjs* that are subsets of Roles, i.e the objects occurring in *D* that model index sets;
DRoles	The union of the members of *DRoSets*, i.e the set of those roles belonging to some member of *DObjs*.

We will add extra primary features, both fixed and variable, to each member of DaMod0 to give us the members of DaMod1.

Each data model is assumed to have a title and suchlike. We model this information without describing it in detail. Each *D* : DaMod1 has the primary variable feature

DModelInfo	The data model's identifier, title, creation date, version number, etc.

Some roles and objects have visible names. We model this with two partial functions :

DRoleName	Partial function assigning visible names to some members of *DRoles*, i.e names for some roles;
DObjName	Partial function assigning visible names to some members of *DObjs*, i.e names for some Entity Types and Fact Types.

We place two restrictions on the names. We require that no two objects have the same name, and that no two roles of the same Fact Type have the same name. In a finished data model we would expect every Entity Type to have a visible name and in some NIAM dialects we would expect every Fact Type to have no more than one unnamed role. However, we wish DaMod1 to model unfinished data models, just as we did for DaMod0, so we do not impose these requirements on the members of DaMod1.

We would expect a data model to include further design information for some roles and objects, though we will not describe it in detail. We model this with two more partial functions :

DRoleInfo	Partial function assigning further design information to some members of *DRoles*, i.e short description, full definition, business rules, etc, for some roles;
DObjInfo	Partial function assigning further design information to some members of *DObjs*, i.e short description, full definition, business rules, etc, for some Entity Types and Fact Types.

Some Entity Types are marked as being Label Types. Recall that a Label Type is an Entity Type whose members are to be represented explicitly in the physical database. Typically they are character strings or numbers. We model the marks with a primary variable feature that says which Entity Types are Label Types :

DLaSets	Subset of *DEnSets* : the Label Types.

Some Fact Types are marked as being derived Fact Types. Recall that the tuples of a derived Fact Type are not added to or removed from the database by the users. Their presence in a database instance is determined by the presence of tuples of other Fact Types. For instance, a pirate's departure date is always his arrival date plus his length of stay. Again, we model the marks with a primary variable feature that says which Fact Types are derived Fact Types :

DDeSets	Subset of *DRoSets* : the index sets identifying the derived Fact Types.

The derivation rules are database constraints. They are modelled later on in Section 7.3.

Defining editor operations for these features would be straightforward. Operations that add or replace a name would need preconditions so that the restrictions are respected.

Extending the editor operations of Chapter 5 so that they apply to the members of DaMod1 would be straightforward, except in the case of the Merge operation. There are preconditions to Merge that ensure that the two members of DaMod0 to be merged are compatible. Additional preconditions are needed for members of DaMod1. For instance, we cannot allow the same role to have two different names. Also, we would expect the result to be given a default title and version number. In practice, the difficult problem is to decide which roles and objects in two data models are to be regarded as the "same". Our model gives no help with this.

7.2 Instances and populations

A database instance is the collection of tuples held in a database at a particular moment in time. In our model the tuples belong to the class Tuple of all fact-style tuples. We have assumed that tuples encoding different kinds of information item have different index sets and so can always be distinguished. Thus our most general model of a database instance is any subset of Tuple.

We have seen that a well-formed NIAM conceptual data model specifies the tuples that are allowed to be held in a database. We have modelled the core part of the data model with some D : DaMod0. Recall that D has the two secondary variable features

DCart	Function assigning a cartesian product to each member of *DObjs*, (the dummy value Φ in the case of Entity Types);
DFacts	The tuples defined by D; the union $\bigcup \{ DCart(t) \mid t : DRoSets \}$.

Thus each possible instance of the database is modelled by some subset of *DFacts*, which, of course, is itself a subset of Tuple. If we wish to determine which tuples of an instance $I \subseteq_d DFacts$ belong to the cartesian product identified by the index set t then we need only form the intersection $I \cap DCart(t)$.

Given any database instance we can extract many different sets of entities and sets of tuples, $I \cap DCart(t)$ above being just one example. We will define a family of **population functions** that extract sets used in the definition of some important kinds of constraint. The terminology varies somewhat among NIAM publications. We will use terms appropriate to our model. We start with a simple example to illustrate these functions, then give the general definitions.

Figure 7.2.1 holds an instance, Ic, of a database that records the enrolment of students in subjects and the examination dates of subjects. The tuples are listed in no particular order. Notice that this may make it difficult to read but it causes no loss of information.

Figure 7.2.1 A database instance, Ic

$$
\begin{array}{ll}
\{ & \{ p \mapsto \text{Carol}, s \mapsto \text{Maths} \}, \\
 & \{ d \mapsto 21.5.98, e \mapsto \text{Geography} \}, \\
 & \{ p \mapsto \text{Carol}, s \mapsto \text{Mechanics} \}, \\
 & \{ p \mapsto \text{Ann}, s \mapsto \text{Physics} \}, \\
 & \{ d \mapsto 25.5.98, e \mapsto \text{Chemistry} \}, \\
 & \{ d \mapsto 21.5.98, e \mapsto \text{Mechanics} \}, \\
 & \{ p \mapsto \text{Carol}, s \mapsto \text{Physics} \}, \\
 & \{ p \mapsto \text{Jim}, s \mapsto \text{Physics} \}, \\
 & \{ p \mapsto \text{Ann}, s \mapsto \text{Maths} \}, \\
 & \{ d \mapsto 22.5.98, e \mapsto \text{Physics} \} \ \}
\end{array}
$$

Given an index set we can determine which tuples of Ic have that index set. The **object population** of the set {d, e} is defined to be the set of tuples of Ic whose index set is {d, e}, namely the set

{ { d $\mapsto$ 21.5.98, e $\mapsto$ Geography },
{ d $\mapsto$ 21.5.98, e $\mapsto$ Mechanics },
{ d $\mapsto$ 22.5.98, e $\mapsto$ Physics },
{ d $\mapsto$ 25.5.98, e $\mapsto$ Chemistry } }.

Given an index we can determine which tuples of Ic use that index, and hence the set of values associated with that index in Ic. The **role population** of the index s is defined to be the set of values associated with the index s in the tuples of Ic, namely the set

{ Maths, Mechanics, Physics }.

Similarly, the role population of the index e is

{ Geography, Mechanics, Physics, Chemistry }.

Given a domain we can determine (from the tuples' domain functions) which tuples of Ic use that domain, hence which indexes are associated with that domain, and hence the set of values belonging to that domain mentioned in the tuples of Ic. The **domain population** of the domain Subject is defined to be the union of the role populations of all roles whose domain is Subject, namely the union of the role populations of s and e, which is the set

{ Maths, Mechanics, Physics, Geography, Chemistry }.

These three functions can be defined for any subset of Tuple, but we might wish to define some more functions that use information taken from the data model. For instance, we might wish to define domain populations ignoring derived Fact Types. Thus we restrict the definitions to the members of DaMod1. For any *D* : DaMod1 we define the secondary variable features

$DPop_O$	Object population function : $DPop_O(I, t) =_d$ the set of tuples of the instance *I* that belong to $DCart(t)$;
$DPop_R$	Role population function : $DPop_R(I, r) =_d$ the set of values (entities or tuples) associated with the index *r* in the tuples of the instance *I*;
$DPop_D$	Domain population function : $DPop_D(I, t) =_d$ the union of the role populations for all roles *r* such that $DConn(r) = t$.

To simplify the definitions, both $DPop_O$ and $DPop_D$ are defined for all members of *DObjs*. If *t* is a set of entities then $DPop_O(I, t)$ is always empty. If *t* is the index set of a Fact Type that is not used as a domain then $DPop_D(I, t)$ is also always empty.

Now we turn to editing operations. The manipulation of database instances is left until Section 7.4. Here we will discuss the effect of changing the data model. Recall from Section 5.2 that if *D* is a member of DaMod0 and *X* is a subset of Objects then $D' =_d$ Tear(D, X) is a member of DaMod0 if a certain amount of care is taken when choosing *X*. Assume that *D'* is a member of DaMod0. Then *D'Objs* is a subset of *DObjs*, and *D'Facts* is a subset of *DFacts*. Given any subset *I* of *DFacts* we can derive a subset $I' =_d I \cap D'Facts$ of *D'Facts*.

We can, if we wish, define *I′* to be the view of the database seen by users whose access is restricted to the part of the database defined by *X*. As the preconditions for *X* are satisfied then the users will see a database that is specified by a well-formed data model.

This construction can be used to define the part of database instances that are to be archived when the database is required to hold archives of "itself". *X* would be chosen to avoid the illogical self-reference described in Section 3.2, Example 11 and Point 17.

Similar constructions can be defined for the Diff and Merge operations. Note, though, that Move can alter cartesian products, rather than just adding or deleting them.

7.3 Constraints

A static database constraint is a declaration that certain instances of the database are acceptable with respect to the constraint, and that all other instances are unacceptable. A **legitimate instance** is one that is acceptable to all the declared static constraints.

A dynamic database constraint is a declaration that certain transitions between instances of the database are acceptable with respect to the constraint, and that all other transitions are unacceptable. A **legitimate transition** is one that is acceptable to all the declared dynamic constraints.

The NIAM guidelines encourage data modellers to declare all known constraints when constructing a conceptual data model. There are several commonly occurring kinds of constraint, and some of these are given special symbols in the pictorial notation. Even where there is no special symbol the constraint's definition is written down, using any suitable notation, and deemed to be part of the data model.

We will add primary features, both fixed and variable, to our extended model to model the various kinds of constraint, including features to model the "unclassified" constraints that have no particular classification. We will model the constraint declarations and also their contributions to the legitimacy or otherwise of database instances and transitions.

The following subsections discuss several different kinds of static constraint, then end with dynamic constraints. Unclassified static constraints illustrate most of the principles and so are discussed first.

7.3.1 Unclassified constraints

An unclassified static constraint is one such as

"No pirate who arrives on a Saturday stays for 5 days"

that fits no particularly common pattern. The only restriction on these constraints is that they should be defined clearly. We will assume that the definitions are well-formed formulas (Wffs) in some language suitable for the purpose. For instance, they might be written in specialised English as above, or in a first order logic language. We model the set of all possible constraint definitions by the set

Wffs	Set of all possible well-formed formulas in any language suitable for talking of data models and the databases they specify.

We model the unclassified constraint declarations in a data model by a subset of Wffs. Each *D* : DaMod1 has the primary variable feature

DConOther	Subset of Wffs : the set of unclassified constraint formulas declared in *D*.

The effect of each static constraint definition, if it is well written, is to flag some database instances as acceptable and the rest as not acceptable. We can characterise each constraint by its set of acceptable instances. We would like to include these sets of instances in our model, but there is a difficulty. Not only are the constraint definitions written in an unspecified language, but the terms in the Wffs refer to the elements of the data model, not to the arbitrary sets used in our model.

We could assume that constraint definitions are written in a particular language which we then model, but this would be presumptuous and not very realistic. Instead, we will assume that each constraint determines some set of acceptable database instances, and then model this set of instances as a primary variable feature. Thus each *D* : DaMod1 has the primary variable feature

DConOtherM	Partial function from *DConOther*; *DConOtherM*(*w*) is (the model of) the set of database instances acceptable to the constraint formula *w*.

We can be sure that given any *DConOther* there is a member of DaMod1 with the right *DConOtherM*. If we are given a particular data model then we can translate its Wffs into a form appropriate to DaMod1, and then obtain a member of DaMod1 where *DConOther* is compatible with *DConOtherM*.

If we collect together the sets of acceptable instances for every static constraint in the data model and form their intersection we obtain the set of instances acceptable to every constraint. That is, we obtain the set of legitimate instances. Each *D* : DaMod1 has the two secondary variable features

DConStaticM	Set of all static constraints, each represented by its set of acceptable instances;
DInstLegits	Set of all legitimate database instances; $\bigcap$ *DConStaticM*.

It is conceivable that there can be constraints that are neither static nor dynamic; for instance, that a database must not alter during working hours. We treat these as unclassified constraints whose effects are not modelled here, which is why *DConOtherM* is a partial function.

Defining editor operations that add and remove unclassified constraints would be straightforward. The extent to which a CASE tool can help to check that the constraints are well defined and reasonable is a non trivial question. We will not attempt to provide an answer here. Note that altering the core part of a data model could render a constraint ill defined.

7.3.2 Implicit constraints

An implicit constraint is one that is assumed to be declared in every NIAM conceptual data model. In effect, it is embedded in the notation.

We will include two implicit static constraints in the extended model. Notice that an implicit constraint is a primary fixed feature and its effect is a secondary variable feature. It cannot be varied independently of other features. We model only the effects of these constraints, expressed as sets of acceptable instances. There is no need to model their definitions as Wffs in some unspecified language.

The first implicit constraint declares that the only tuples allowed in the database are those defined by the data model. This might seem redundant but it is included for two reasons. First, it is a key principle of the NIAM design technique : a data model states what may be recorded in the database. Second, it ensures that the intersection used to define legitimate instances is always well defined. Recall for each D : DaMod0 that $DFacts$ is the union of all the Fact Types defined by D. An instance acceptable to this outermost constraint is any subset of $DFacts$. Thus each D : DaMod1 has the secondary variable feature

DConOuterM	Implicit outermost constraint, modelled as a set of acceptable instances; $DConOuterM =_d \mathsf{Pow}(DFacts)$.

The second implicit constraint concerns objectified Fact Types. Recall that one tuple can be an element of another tuple. For instance, we might have a tuple saying that Carol studies Physics, and another tuple saying that this combination of student and subject got a mark of 73. We would not allow this mark to be recorded if there were no record that Carol studies Physics. The general rule is that a tuple with elements that are tuples is not allowed in the database unless its element tuples are also present.

We can express this rule using the population functions of Section 7.2. In any database instance the tuples that are elements belong to domain populations and the tuples that are present belong to object populations. The instance is acceptable only if each Fact Type's domain population is a subset of its object population. In practice this is a restriction on data modellers. They must not show one Fact Type as being the domain of another Fact Type unless this constraint is appropriate.

As usual, we model this constraint as a set of acceptable instances. Each D : DaMod1 has the secondary variable feature

DConObM	Implicit objectified Fact Type constraint, modelled as a set of acceptable instances.

Its definition is

$$DConObM =_d \{ I \subseteq_d DFacts \mid \forall t : DRoSets \bullet DPop_D(I, t) \subseteq DPop_O(I, t) \}.$$

As *DConOuterM* and *DConObM* are secondary features there are no editing operations that act directly on them, and they do not impose any additional constraints on other editing operations.

7.3.3 Derived Fact Types

A **derived Fact Type** is one whose object population is determined by the object populations of other Fact Types in all legitimate instances. For instance, in the Plunder Inn register a pirate's departure date is always his arrival date plus his length of stay.

A data model marks some Fact Types as being derived Fact Types. This was modelled in Section 7.1. A finished data model will also include a definition for each derived Fact Type. As usual, we will allow data models to be unfinished, with some definitions absent.

The definitions will be treated in the same way as the unclassified constraints. We will assume that the definitions are well-formed formulas in some language suitable for the purpose. We will model them with a partial function from the index sets identifying the derived Fact Types to the set Wffs that we defined earlier. Each *D* : DaMod1 has the primary variable feature

DConDerived	Partial function from *DDeSets* to Wffs; *DConDerived*(*t*) is the formula defining the (evolving) object population of the derived Fact Type whose index set is *t*.

We will model the effect of each definition as a set of acceptable database instances as we did for the unclassified constraints. Each *D* : DaMod1 has the primary variable feature

DConOtherM	Partial function from *DDeSets*; *DConOtherM*(*t*) is (the model of) the set of acceptable database instances defined for the derived Fact Type whose index set is *t*.

We require that a set of instances is assigned to each member of *DDeSets* that has a Wff assigned to it. We also require that only the appropriate object population is constrained in each set of instances.

The remarks on editor operations for unclassified constraints also apply here.

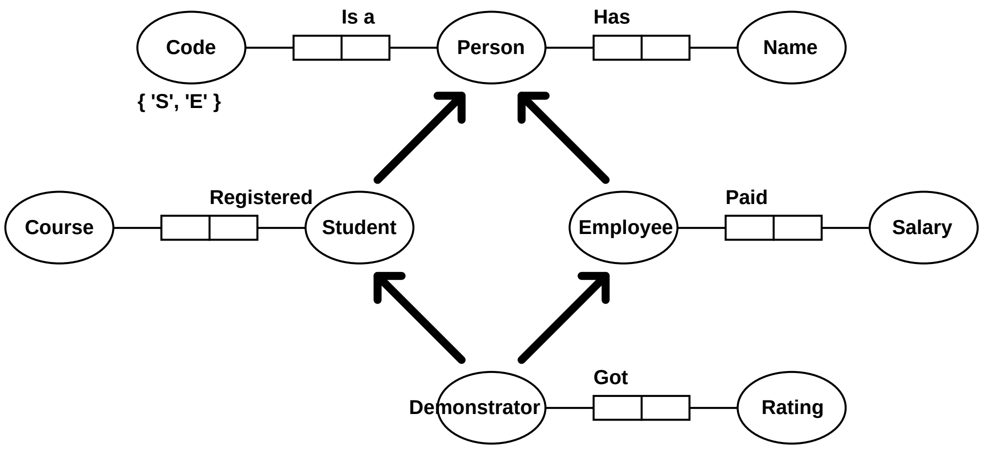
7.3.4 Subtypes

A **Subtype constraint** declares that certain role populations must be restricted to those members of a domain population that obey a given rule. For instance, whereas a salary is recorded for every employee, a bonus can be recorded only for employees flagged as management grade. A Subtype, such as all management grade employees, is best thought of as an evolving subset of an evolving domain population. Subtype constraints allow data models to talk of evolving domains and overlapping domains without violating the principle that Entity Types are fixed and pairwise disjoint.

There are several ways that Subtype constraints could be described in the extended model. The simplest way would be to mark those roles subject to a Subtype constraint and provide a function assigning each marked role to its constraining formula, and also a function assigning a name to each formula.

However, the various NIAM pictorial notations use a symbol resembling the Entity Type symbol to display Subtypes. An example, using the main NIAM notation, is shown in Figure 7.3.4.1 below. This data model says that any person can have a name, but only people flagged as Students can register for a course and only people flagged as employees can get paid. Also, only people who are both can be given a demonstrator rating.

Figure 7.3.4.1 A data model with Subtype symbols (using the main NIAM dialect)

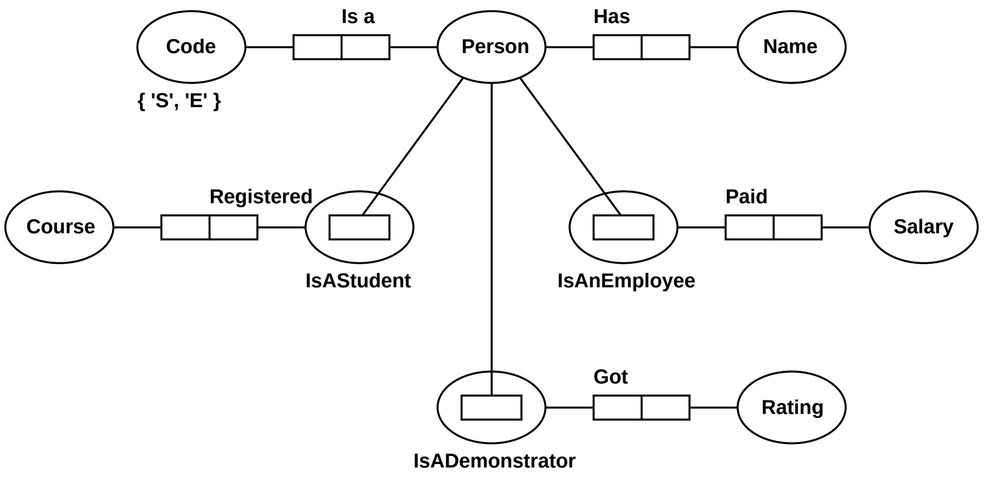


We should assume that the diagram contains a line from the role Registered to its domain Person but that the line is partly overlaid by the heavy arrow, which indicates a Subtype constraint, and by the ellipse, which gives a name to the evolving subset.

We wish the extended model to have components that correspond to part of the Subtype notation. We will not model all the features of the notation here. We do this by giving a different, but equivalent, meaning to Subtype constraints.

We can enhance the database specified in Figure 7.3.4.1 so that it keeps a (redundant) record of the people who are students. We add the derived unary Fact Type IsAStudent to do this. Likewise we add the Fact Types IsAnEmployee and IsADemonstrator to record those who are employees and those who are student employees respectively. Now we move the role Registered so that its domain is the Fact Type IsAStudent. We do likewise for the roles Paid and Got, giving us the data model shown in Figure 7.3.4.2.

Figure 7.3.4.2 Subtypes replaced by unary Fact Types



We now have a data model whose core part is modelled by a member of DaMod0, containing objects representing each Subtype symbol. Observe that the registration Fact Types in the two data models are equivalent in the FlatEq sense of Section 5.5. Furthermore, for any person *x*, by the implicit objectified Fact Type constraint of

Section 7.3.2, IsAStudent(*x*) can be registered for a course in the modified database iff *x* can be registered in the original database. The two databases are equivalent. In practice, data models are flattened before being transformed into database schemas so the two databases will be identical.

This technique for representing Subtypes by derived unary Fact Types can be applied to any data model. We will include features in our extended model that mark some Fact Types as representing Subtypes and that assign formulas to these Fact Types. Subtype names will be represented by Fact Type names, which are already modelled. Each *D* : DaMod1 has the two primary variable features

DSuSets	Subset of *DRoSets* : the index sets identifying the derived unary Fact Types that represent Subtypes;
DConSubtype	Partial function from *DSuSets* to Wffs : *DConSubtype*(*t*) $=_d$ the formula defining the Subtype represented by the Fact Type whose index set is *t*.

We require that marked Fact Types are unary. Although these Fact Types are a special kind of derived Fact Type we will keep them separate by requiring that no Fact Type is marked as both derived and Subtype.

We model the effects of Subtype constraints in the usual way. Each *D* : DaMod1 has the primary variable feature

DConSubtypeM	Partial function from *DSuSets* : *DConSubtypeM*(*t*) $=_d$ (the model of) the set of acceptable instances for the Subtype constraint represented by the Fact Type whose index set is *t*.

As with derived Fact Types we require that a set of acceptable instances is assigned to each member of *DSuSets* that has a Wff assigned to it and that only the appropriate object population is constrained in each set of instances. We also require that each domain population is entirely determined by the role populations of the roles not used in representing Subtypes. This is equivalent to the rule that a Subtype constraint must not be defined in terms of itself. For instance, in Figure 7.3.4.1 one constraint says that only employees get paid, so we must not define employees as those people who get paid.

Notice that we allow Fact Types to have Subtypes. This may be unusual but is entirely legitimate. We can talk of appointments that are flagged as urgent regardless of whether appointments are modelled by an Entity Type or a Fact Type.

We can define another population function. A **Subtype population** is the set of those members of a domain population that satisfy the Subtype's defining formula. We can include a subtype population function in the extended model but its values must be derived from the population of the Fact Type representing the Subtype instead of from the defining formula. Each *D* : DaMod1 has the secondary variable feature

$DPop_S$	Subtype population function : $DPop_S(I, t) =_d$ the Subtype population of the Subtype represented by the Fact Type whose index set is *t*, in the instance *I*; $t = \{r\}$ for some role *r*, and $DPop_S(I, t) =_d$ the role population $DPop_R(I, r)$.

The remarks on editor operations for unclassified constraints also apply here.

The Subtype notation includes arrows and other symbols which we do not model here. These symbols give the reader information about the properties of the defining formulas. For instance, the arrows in Figure 7.3.4.1 state that every demonstrator is both a student and an employee. There are symbols that could tell us that in legitimate instances every member of Person's domain population must be a student or an employee or both. These symbols are theorems deducible from the Subtype definitions, written in an unusual language.

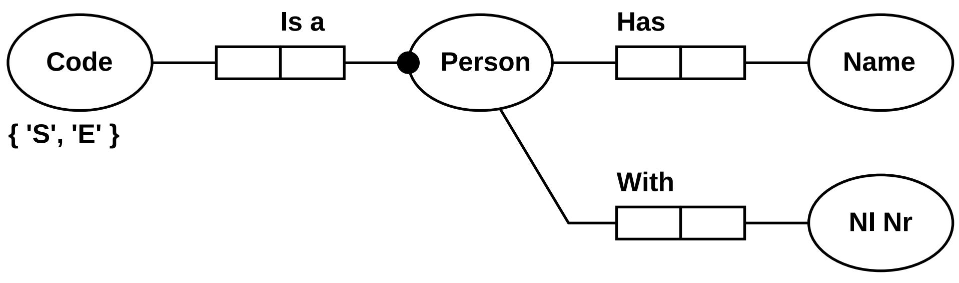
It can be shown that the Subtype arrows in any well-formed data model can always be modelled as the graph of a finite Well Founded relation. (It is acyclic and anti-reflexive). Consequently the arrows have the same structure as the connections in some member of DaMod0. Anyone given the task of implementing a means of describing the Subtype arrows should consider re-using the technology used to describe the core part of data models, including the editing operations.

7.3.5 Diagram symbols

The NIAM pictorial notation uses symbols to declare the more common kinds of constraint without the need to write down any defining formulas. We will include two of these symbols in the extended model to illustrate the principles. Notice that if we model a constraint symbol we can model its set of acceptable instances exactly; there are no Wffs in an unknown language to be translated here. The elements in the extended model that represent these symbols will be called specifiers, as each specifies that a particular constraint has been declared.

The first symbol we model is used to declare **total role constraints**. An example is given in Figure 7.3.5.1. The blob on the line from the role "Is a" to the Entity Type Person indicates that if any person's name or NI number is recorded then their classification code, S or E or both, must also be recorded.

Figure 7.3.5.1 A total role constraint



In general, if a role is subject to a total role constraint then its role population must equal the domain population of its domain for a database instance to be acceptable. We model the blobs by marking those roles subject to a total role constraint. Each *D* : DaMod1 has the primary variable feature

DConTotal	Subset of *DRoles* : the specifiers of the total role constraints.

We define a family, *DConSpec*, of constraint specification functions. Each function returns the set of acceptable instances when given a specifier of the appropriate kind.

Here, we define the constraint specification function for total role constraints. Each D : DaMod1 has the secondary variable feature

$DConSpec_T$	Total function on *DRoles* : $DConSpec_T(r) =_d$ the set of instances acceptable to a total role constraint on the role *r*.

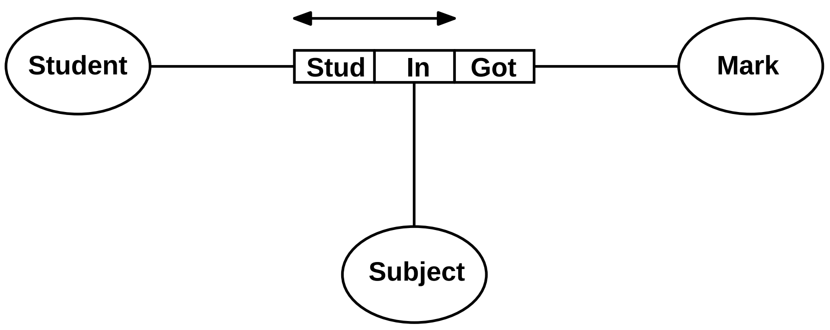
Its definition is

$$DConSpec_T(r) =_d \{ I \subseteq_d DFacts \mid DPop_R(I, r) = DPop_D(I, DConn(r)) \}.$$

We apply $DConSpec_T$ to each role belonging to *DConTotal* to give us the set of acceptable instances for each total role constraint declared in *D*.

The second symbol we model is used to declare One-Fact-Type **uniqueness constraints**. (In the literature they are variously called intra-fact and internal uniqueness constraints). An example is given in Figure 7.3.5.2. The double-headed arrow over the roles Stud and In indicates that at most one mark is given to each student in each subject. In different words, each combination of a student and a subject occurs at most once.

Figure 7.3.5.2 A Uniqueness constraint on one Fact Type



In general, an arrow covers one or more roles of a Fact Type. A database instance is acceptable iff it does not contain different tuples having the same values at each covered role. We model each arrow by the set of roles it covers. Each D : DaMod1 has the primary variable feature

DConUnique1	Set of subsets of *DRoles* : the specifiers of the One-Fact-Type uniqueness constraints.

We require that each member of *DConUnique1* is a non-empty subset of one index set. (Other kinds of uniqueness constraint can span several index sets).

We define a constraint specification function for this kind of constraint. Each D : DaMod1 has the secondary variable feature

$DConSpec_{U1}$	Total function on the subsets of *DRoles* : $DConSpec_{U1}(R) =_d$ the set of instances acceptable to a One-Fact-Type uniqueness constraint on the set *R* of indexes.

Its definition is

$$DConSpec_{U1}(R) =_d$$

$$\{\ I \subseteq_d DFacts \mid \forall T1, T2 : I \bullet (\ R \subseteq T1I\ \wedge\ R \subseteq T2I\ \wedge\ R\langle T1Val = R\langle T2Val\ )\ \Rightarrow\ T1 = T2\ \}.$$

Defining editor operations that add or remove constraint specifiers would be straightforward. Extending the operations of Chapter 5 for these constraints would also be straightforward. Several tests for the plausibility of these constraints or combinations of constraints are described in the literature. Some can be applied mechanically by a CASE tool. For instance, a One-Fact-Type uniqueness constraint should not leave more than one role uncovered.

7.3.6 Dynamic constraints

It appears that dynamic constraints are not classified in any way in NIAM. We will model them as simply as possible : as a set of constraint definitions and a set of all legitimate transitions. Each *D* : DaMod1 has the two primary variable features

DConDynamic	Subset of Wffs : the set of dynamic constraint formulas declared in *D*;
DTransLegits	The set of all legitimate database transitions.

7.4 Updates and queries

The construction and use of databases are outside the scope of this work but we should confirm that our model can be used to describe database operations if desired.

Assume from now on that the database's specification is modelled by a member *D* of DaMod1. We know that each database instance is modelled by a subset of *DFacts*. The most primitive operations to alter the database contents are

a) Empty the database;

b) Insert one tuple;

c) Remove one tuple.

Clearly, we can model these operations in a straightforward way. For instance, the result of operation (a) is the instance $\varnothing$.

However, it is desirable that users see only legitimate instances and we may need to insert and remove several tuples to go from one legitimate instance to another. This naturally leads to the well known concept of a **transaction**, in which several tuples are inserted and removed in a process whose details are not visible to users. We can model transactions by an operator that is given an instance $I \subseteq_d DFacts$, a set $In \subseteq_d DFacts$ of tuples to insert, and a set $Re \subseteq_d DFacts$ of tuples to remove, and returns the updated instance $I \cup In \setminus Re$. If *In* and *Re* are disjoint then there is no ambiguity about the desired result. Clearly, any transition from one instance to another can be modelled as an application of this operator.

Thus database updates can be modelled in a simple and natural way. Notice that the model makes no assumptions about the internal organisation of the database.

A query can be described as a function that can be applied to any database instance to give a result. The result need not be a set of tuples. It could be a single value of any kind. Obviously, if the function can be defined then it can be modelled.

Alternatively, we may wish to define a meta-function which, on being given the definition of a query, returns the appropriate result. The query definitions would be expressions in a particular query language. Obviously, this too can be modelled if the semantics of the query language are clear. Note, though, that if the query language is too expressive then some query operations might not terminate.

There seems to be no barrier to modelling queries. However, popular query languages such as SQL act on encoded database instances. A general model of un-encoded instances will not always be the best vehicle for investigating the behaviour of query processors.

7.5 Details

In this section we define the features of each member of DaMod1, the extended model of all NIAM conceptual data models. The definition includes some ellipsis symbols to show where additional features could be added, for instance to describe more kinds of constraint symbols.

We start by introducing some constants used to model three kinds of text annotation.

Constants

Some constants used in the definition of DaMod1

Wffs : Set	All Well Formed Formulas in any language appropriate to data models
Names : Set	All possible names
Infos : Set	All composite information (not defined in detail)

Now we define DaMod1. The definition is split into three definition blocks. The first introduces most of the primary and secondary features. It contains forward references to the second block, which defines population functions (*DPop*), and to the third block, which defines constraint specification functions (*DConSpec*).

Some reminders may be helpful. The image function [[ ]] was defined in Section 5.4.2. The constraint Wffs declared in a data model are modelled separately from their effects, which are modelled as sets of acceptable instances. Finally, for any *D* : DaMod0

DFacts	is the set of all permitted tuples;
Pow(*DFacts*)	is the set of all possible database instances;
Pow(Pow(*DFacts*))	is the set of all possible sets of database instances.

D : DaMod1

The class of all models of well-formed, but not necessarily finished, NIAM conceptual data models. (DaMod0 extended to include annotation).

∗ *D*[DaMod0] : DaMod0	Core data model (with features *DObjs*, *DConn*, *DRoles*, *DCart*, *DFacts*, etc.)
[General information]	
∗ *DModelInfo* : Infos	*D*'s model identifier, title, creation date, version identifier, etc.

[Names]

∗	$DRoleName : DRoles +\!\!\rightarrow \mathrm{Names}$	Visible role names
∗	$DObjName : DObjs +\!\!\rightarrow \mathrm{Names}$	Visible Entity Type & Fact Type names

$DCoNam1$.: $\forall R : DRoSets \bullet \mathrm{IsInjection}(R \langle DRoleName)$

$DCoNam2$.: $\mathrm{IsInjection}(DObjName)$

[Design annotation]

∗	$DRoleInfo : DRoles +\!\!\rightarrow \mathrm{Infos}$	Short description, full definition, business rules, etc.
∗	$DObjInfo : DObjs +\!\!\rightarrow \mathrm{Infos}$	Short description, full definition, business rules, etc.

[Markings]

∗	$DLaSets \subseteq_{\mathrm{d}} DEnSets$	The Label Types
∗	$DDeSets \subseteq_{\mathrm{d}} DRoSets$	(The index sets of) the derived Fact Types
∗	$DSuSets \subseteq_{\mathrm{d}} DRoSets$	(The index sets of) the Subtypes (each modelled by a unary Fact Type)

$DCoSu1$.: $\forall R : DSuSets \bullet DArity(R) = 1$

$DCoSu2$.: $DSuSets \cap DDeSets = \varnothing$

[Static constraints]

[Static : Constraint Wffs occurring in the data model]

∗	$DConDerived : DDeSets +\!\!\rightarrow \mathrm{Wffs}$	Derived Fact Type definitions
∗	$DConSubtype : DSuSets +\!\!\rightarrow \mathrm{Wffs}$	Subtype definitions
∗	$DConOther \subseteq_{\mathrm{d}} \mathrm{Wffs}$	Unclassified constraint formulas. Note : some might not be static constraints.

[Static : Implicit constraints]

$DConOuterM =_{\mathrm{d}} \mathrm{Pow}(DFacts)$	Implicit outermost constraint, modelled as a set of acceptable instances. All database instances must be subsets of *DFacts*.

$DConObM =_{\mathrm{d}} \{\, I \subseteq_{\mathrm{d}} DFacts \mid \forall t : DRoSets \bullet DPop_{\mathrm{D}}(I, t) \subseteq DPop_{\mathrm{O}}(I, t) \,\}$

Implicit objectified Fact Type constraint, modelled as a set of acceptable instances. Only tuples recorded in the database can play roles.

[Static : Model of (the effect of) constraint Wffs]

∗ $DConDerivedM : DDeSets \mathrel{+\!\!\rightarrow} \text{Pow}(\text{Pow}(DFacts))$ — Derived Fact Type definitions, each modelled as a set of acceptable instances

$DCoDe1$.: $DConDerivedMDef = DConDerivedDef$

$DCoDe2$.: Each definition restricts only one Fact Type population
$\forall t : DConDerivedMDef \bullet$
$\{ I \setminus DCart(t) \mid I : DConDerivedM(t) \} = \text{Pow}(DFacts \setminus DCart(t))$

∗ $DConSubtypeM : DSuSets \mathrel{+\!\!\rightarrow} \text{Pow}(\text{Pow}(DFacts))$ — Subtype definitions, each modelled as a set of acceptable instances

$DCoSu3$.: $DConSubtypeMDef = DConSubtypeDef$

$DCoSu4$.: Each definition restricts only one Fact Type population
$\forall t : DConSubtypeMDef \bullet$
$\{ I \setminus DCart(t) \mid I : DConSubtypeM(t) \} = \text{Pow}(DFacts \setminus DCart(t))$

$DCoSu5$.: Each Subtype population is a subset of its domain's non-subtype population
$\forall t : DConSubtypeMDef \bullet \forall I : DConSubtypeM(t) \bullet \forall r : t \bullet$
$DPop_R(I, r) \subseteq$
$\{ TVal(r') \mid T : I \wedge TI \notin DSuSets \wedge r' : TI \wedge DConn(r') = DConn(r) \}$

∗ $DConOtherM : DConOther \mathrel{+\!\!\rightarrow} \text{Pow}(\text{Pow}(DFacts))$ — Effect of unclassified static constraint formulas, each modelled as a set of acceptable instances

[Static : Constraint specifiers]

∗ $DConTotal \subseteq_d DRoles$	Specifiers of total role constraints
∗ $DConUnique1 \subseteq_d \text{Pow}(DRoles)$	Specifiers of One-Fact-Type uniqueness constraints

$DCoU1$.: $\forall R : DConUnique1 \bullet R \neq \varnothing \wedge \exists t : DObjs \bullet R \subseteq t$

∗ …	Specifiers of any more kinds of constraint

[Static : Overall]

$DConStaticM \subseteq_d \text{Pow}(\text{Pow}(DFacts))$	All static constraints, each represented by its set of acceptable instances

$DDefConStaticM$.:
$DConStaticM =_d$
$\{ DConOuterM, DConObM \} \cup$
$DConDerivedMRan \cup DConSubtypeMRan \cup DConOtherMRan \cup$
$DConSpec_T[\![DConTotal]\!] \cup$
$DConSpec_{U1}[\![DConUnique1]\!] \cup$
… (any more kinds of constraint defined by specifiers)

	$\mathit{DInstLegits} =_d \bigcap \mathit{DConStaticM}$	All legitimate database instances
	[Dynamic constraints]	
	[Dynamic : Constraint Wffs in the data model]	
∗	$\mathit{DConDynamic} \subseteq_d$ Wffs	All dynamic constraints
	[Dynamic : Model of (the effect of) constraint Wffs]	
∗	$\mathit{DTransLegits} \subseteq_d \mathrm{Pow}(\mathit{DFacts}) \times \mathrm{Pow}(\mathit{DFacts})$	All legitimate database transitions
	[etc]	
∗	…	More features, e.g triggers

D : DaMod1 Secondary features : 1

Some features of each member of DaMod1 : Population functions

$DPop$	Family of population functions,
$DPop_{\alpha}$: Function	defined below
(α : { O, R, D, S, … })	

$DPop_{O}$	Object population function
$DPop_{O}(I, t) \subseteq_{d} DFacts$	Object population of object t in instance I
($I \subseteq_{d} DFacts$, $t : DObjs$)	(= ∅ for Entity Types)

$DDefPop_{O}$.: $\forall I \subseteq_{d} DFacts \bullet \forall t : DObjs \bullet$
$DPop_{O}(I, t) =_{d} I \cap DCart(t)$

$DPop_{R}$	Role population function
$DPop_{R}(I, r) \subseteq_{d} (DEntities \cup DFacts)$	Population of role r in instance I
($I \subseteq_{d} DFacts$, $r : DRoles$)	

$DDefPop_{R}$.: $\forall I \subseteq_{d} DFacts \bullet \forall r : DRoles \bullet$
$DPop_{R}(I, r) =_{d} \{ TVal(r) \mid T : I \wedge r \in TI \}$

$DPop_{D}$	Domain population function
$DPop_{D}(I, t) \subseteq_{d} (DEntities \cup DFacts)$	Domain population of object t in
($I \subseteq_{d} DFacts$, $t : DObjs$)	instance I. (= ∅ if $t \notin DConnRan$)

$DDefPop_{D}$.: $\forall I \subseteq_{d} DFacts \bullet \forall t : DObjs \bullet$
$DPop_{D}(I, t) =_{d} \{ TVal(r) \mid T : I \wedge r : TI \wedge DConn(r) = t \}$

$DPop_{S}$	Subtype population function
$DPop_{S}(I, t) \subseteq_{d} (DEntities \cup DFacts)$	Subtype population for Subtype t
($I \subseteq_{d} DFacts$, $t : DSuSets$)	in instance I

$DDefPop_{S}$.: $\forall I \subseteq_{d} DFacts \bullet \forall t : DSuSets \bullet$
$\forall r : t \bullet DPop_{S}(I, t) =_{d} DPop_{R}(I, r)$

…	More population functions

D : DaMod1 Secondary features : 2

Some features of each member of DaMod1 : Constraint specification functions

$DConSpec$ $DConSpec_{\alpha}$: Function (α : { T, U1, … })	Family of constraint specification functions, defined below
$DConSpec_{T}$ $DConSpec_{T}(r) \subseteq_{d}$ Pow($DFacts$) (r : $DRoles$)	Total role constraint Constraint for the role *r*, modelled as a set of acceptable instances

$DDefConSpec_{T}$.: $\forall r : DRoles \;\bullet$
$DConSpec_{T}(r) =_{d} \{ \mathrm{I} \subseteq_{d} DFacts \mid DPop_{R}(I, r) = DPop_{D}(I, DConn(r)) \}$

$DConSpec_{U1}$ $DConSpec_{U1}(R) \subseteq_{d}$ Pow($DFacts$) ($R \subseteq_{d} DRoles$)	One-Fact-Type uniqueness constraint Uniqueness on the roles *R*, modelled as a set of acceptable instances

$DDefConSpec_{U1}$.: $\forall R \subseteq_{d} DRoles \;\bullet$
  $DConSpec_{U1}(R) =_{d}$
    $\{ \mathrm{I} \subseteq_{d} DFacts \mid$
      $\forall T1, T2 : I \;\bullet$
        $(R \subseteq T1I \;\wedge\; R \subseteq T2I \;\wedge\; R\langle T1Val = R\langle T2Val) \;\Rightarrow\; T1 = T2 \}$

…	More constraint specification functions

8 Conclusions

We now have sufficient information to answer the questions posed in Section 1.1. We end this work with the answers and some additional observations.

We start with a brief reminder of what has been done in Chapters 3 to 7 (Section 8.1). We then answer the questions (Section 8.2). Recall that the general theme of the questions was

"When can I use NIAM and how might design tools help me?".

Further observations are appropriate, some on data modelling (Section 8.3) and some on modelling in general (Section 8.4). The latter includes a comparison between the model used here and others that have been published, and some heuristics for building mathematical models.

We finish, of course, by listing some topics that deserve further investigation (Section 8.5).

8.1 Overview

Before answering the questions posed in Section 1.1 let us remind ourselves of some of the key ideas and results in Chapters 3 to 7.

We saw in Chapter 3 that the word "database" covers a larger class of objects than we might have expected. The primary objective of a database is to hold information that users wish to be reminded of. It matters not whether the information is recorded on computer discs, paper, clay tablets, or even in museum display cabinets.

We saw that each item of information in a database can be described as having a fixed part, common to several items, and a variable part peculiar to itself. The variable part can be represented as a tuple in which there is a set of indexes with a value attached to each index. The indexes form a link between the fixed part and the values, enabling us to reconstitute the full information item. Thus databases can be modelled as evolving sets of tuples (and are often implemented this way).

We saw that the contents of some databases are restricted. The purpose of a NIAM conceptual data model is to specify the permitted contents of a database. The core part of a well-formed data model defines a set of tuples in the form of a set of cartesian products, alias Fact Types. The non-core part includes constraint declarations that define some subsets of this set of tuples to be legitimate database instances.

We saw in Chapter 4 that each member of the set DaMod0 not only models the core part of a data model but also defines a uniquely determined set of cartesian products. Thus DaMod0 models well-formed core data models. And in Chapter 6 we saw that DaMod0 models all well-formed core data models, with a reasonable definition of "all".

We saw in Chapter 5 that we can define a small set of functions that act as a basis for any editing operations likely to be needed. For each function there is a simple pre-condition ensuring that the result of altering a well-formed data model is still well-formed.

We noted that the NIAM literature describes many cases of equivalent constructions : different ways of meeting the same user requirements. We saw that another case can be defined. It is based on the flattening of Fact Types whose domains are also Fact Types. A simple test for this kind of equivalence can be applied to Fact Types nested to any degree.

Finally, we saw in Chapter 7 that the model of core data models can be extended to model the non-core parts in a straightforward and natural way.

8.2 NIAM properties

The questions posed in Section 1.1 are answered in the following subsections. Each subsection is devoted to a single question.

8.2.1 What does NIAM do?

Question 1

What, if anything, does a conceptual data model prescribe?

A well-formed NIAM conceptual data model defines a uniquely determined set of Fact Types, alias cartesian products. Any database specified by the data model must be capable of holding any tuple of these Fact Types, and no other tuples are allowed. Typically, the data model will also restrict the permitted evolution of the database. In particular, only certain combinations of tuples are allowed : the legitimate database instances.

The data model specifies a "conceptual" database, and it declares that only tuples of certain Fact Types are to be recorded explicitly (the "actual" database). The presence of any other tuples in the database is to be deduced by users.

The Entity Type symbols, the Fact Type symbols, and the lines joining them describe the core part of a NIAM conceptual data model. The core part is well formed if it is modelled by some member of DaMod0. In outline, this will be so if the data model can be constructed by starting with the empty data model and adding one Entity Type or Fact Type at a time in a sensible way. In particular, when a Fact Type is added its domains must be Entity Types or Fact Types added earlier. (The detailed requirements are modelled by the generators EmpDaMod0, AddedEn, and AddedRo).

The non-core part is well formed if annotation has been attached to the core part in a sensible way, and also any constraint formulas are well formed and appropriate, a topic that has not been investigated in detail in this work.

8.2.2 Implementation work

Question 2

Can the rest of the development work on a database be classified as "implementation" : deciding *how* rather than *what*?

It is obvious from the answer to question 1 that any further work can properly be classified as implementation. However, there are two topics that should be discussed in more detail.

First, the choice of some Entity Types may be deferred until the implementation stage. For instance, the precise range of numbers to be implemented might not be known until the database management system is chosen. This is reasonable.

Second, the Fact Types defining the "actual" database are seldom shown in data models. As there is an algorithm for generating these Fact Types, for instance the Rmap algorithm in Halpin [1995], this is also reasonable.

8.2.3 When can NIAM be used?

Question 3

What simple test, if any, will recognise when NIAM can be used; and when it cannot?

There are some circumstances where we do not know whether or not the NIAM design technique can be used. However, these circumstances can be described and are unlikely to arise in a conventional office or industrial information system. Thus we can construct a useful test that gives the answer Yes, No, or (occasionally) Unknown, as follows :

a) If all of the items Y1 to Y6 below are true of the product to be designed, and none of U1 to U3, then NIAM can be used.

b) If any of items Y1 to Y6 are false then NIAM cannot be used, or would be inappropriate.

c) If all of items Y1 to Y6 are true but U1, U2, or U3 is also true then we do not know whether NIAM can be used. Further analysis of the product and of design techniques would be needed.

The following items are pre-requisites for the use of NIAM.

Y1 The object to be designed is a database, in the widest possible sense.

Section 3.1.1 lists some examples of objects that can be treated as databases.

Y2 The possible contents of the database are to be prescribed in advance.

Section 3.2.1 gives some reasons why the contents might be prescribed in advance.

Y3 The description of the possible contents of the database changes infrequently, and normal users cannot change this description.

Note that even if the description changes frequently then the description and the tuples it describes might be held in a meta-database whose description does not change.

Y4 Entity Types with fixed memberships are a reasonable approximation to reality.

For instance, when designing a register of cars we can reasonably talk of all possible cars : past, present, and future. Note that Subtypes can be used to give the appearance of evolving Entity Types.

Y5 The number of Fact Types is finite, and not unmanageably large. Each Fact Type has a finite arity.

Y6 It is reasonable to represent items of information as tuples.

Section 3.1.2 shows how this can be done. Note that the database implementation can encode these tuples in any way desired.

The following items describe circumstances outside the scope of this work.

U1 Tuples are allowed to have circular definitions, either because they have an unusual construction rule or a set theory lacking the axiom of foundation is in use; e.g a tuple can be an element of itself.

Then the meaning of data models and databases must be re-investigated.

U2 Some Entity Type is defined to be a mathematical object that cannot be modelled by a set; e.g a proper class.

Then ways to describe the association of roles with domains must be re-investigated.

U3 It is sometimes impossible to say whether or not a particular tuple is held in a database instance; e.g only a probability can be ascertained.

Then the meaning of data models and databases must be re-investigated.

8.2.4 Proper operations

Question 4
What are the proper operations for constructing NIAM conceptual data models, for altering them, and for re-using parts in other projects?

Many proper operations can be defined. Chapter 5 contains the definitions of some editing functions acting on the core parts of data models. Any others likely to be needed can be composed from these. Each of them has a simple precondition that ensures that the result of operating on a well formed core data model is also well formed.

Extending operations to the non-core parts of data models has not been studied in detail. However, it is clear from the outline in Chapter 7 that defining proper operations would be straightforward, if not trivial, for the most part. The only difficulty is to ensure that constraint formulas such as those defining Subtypes are well-formed and appropriate.

8.2.5 Helpful design tools

Question 5
Can computerised design tools facilitate all reasonable operations while precluding improper ones?

Clearly, the editing functions defined in Chapter 5 can be implemented by a design tool and the preconditions enforced. Thus a design tool can ensure that an evolving data model always has a well-formed core part. Provided the answer to the test in Section 8.2.3 is Yes then these operations allow every possible well-formed core data model to be constructed, subject only to storage limits.

Equally clearly, a test for equivalence in the FlatEq sense defined in Chapter 5 can also be implemented by a design tool. We have shown that there is a simple algorithm for doing this test.

From the outline in Chapter 7 it is clear that a design tool can implement all reasonable operations on the non-core part of an evolving data model. It can enforce many plausibility restrictions in a helpful way but there will be some things it cannot check. For instance, if constraint formulas can be written in any arbitrary language then it cannot ensure that the formulas are well-formed.

8.2.6 Design tool essentials

Question 6

What are the essential components and structures that any computerised design tool must implement?

The definition of PreMod in Chapter 4 specifies the minimum requirements for any implementation of the core part of NIAM conceptual data models. If we apply the test in Section 8.2.3 to the design tool we get the answer Yes, so NIAM is a candidate design technique for the storage part of the design tool. Section 4.5.2 translates the definition of PreMod into a data model specifying a practical database for holding one core data model. Notice that the proper operations translate into dynamic constraints on this database.

The definition of DaMod1 in Chapter 7 is an incomplete specification of the extra requirements for implementing entire data models.

8.3 Observations on data modelling

There are some observations about conceptual data modelling that deserve to be made.

Observation 1
The definitions of "role", "entity", and "label" vary from publication to publication. See, for instance, the definitions reported in Sections 2.1.1.1 and 2.2.2. This must surely be confusing to students. The definitions developed in this work seem to be straightforward and have worked well.

Observation 2
We must always be clear whether

a) A database specification assumes fixed domains or variable domains;

b) A definition refers to fixed domains or evolving populations;

c) A rule constrains the contents of the database or is a business rule constraining the users.

Observation 3
There appears to be no way to summarise a data model first and then fill in the details later. Thus top-down design does not appear to be possible. However, there are two well-known ways that allow us to concentrate on part of a data model :

a) Build several data models separately, then merge them in a view integration process;

b) Leave out Label Types, and the Fact Types that use them, in the early stages of the design.

8.4 Observations on modelling

There are also some observations about modelling in general to be made.

8.4.1 General observations

Observation 1
Scheurer's Feature Notation has worked well for defining classes, for defining operations, and in proofs.

Observation 2
When modelling a design process we must avoid confusion over the meaning of "well-formed". One meaning is that the design is finished and is ready for mass production or implementation, as the case may be. Another meaning is that the design may be incomplete but is in a satisfactory state. These are different meanings, with different definitions!

Observation 3
In Section 5.4 we defined a base conversion operation and used it to define an isomorphism (BaseEq). In effect, we declared that the things preserved by this isomorphism are significant, and that the things not preserved are mere implementation details. When building any model we should consider defining such an isomorphism, especially where it is possible to define several different isomorphisms.

Observation 4
If the objects we are modelling all have construction sequences then we should consider modelling them as an inductively generated set. If the objects we are modelling all have a Well Founded relation as a feature then there may be construction sequences that we can usefully exploit. We should notice that construction sequences imply the existence of an incremental, systematic, development process.

Observation 5
It is surprising that Well Founded relations and their recursion theorem (Section 4.3.2) are not better known. After all, any computer program that has functions calling more primitive functions is making practical use of the theorem. Modellers should be aware of the general rule.

Observation 6
We speculated in Section 2.3.3.1 that NIAM data models might belong to Geoffrion's very general class of definitional systems. Recall that a definitional system defines a set of objects, where each object is either a primitive object defined elsewhere or a complex object constructed from primitive objects and less complex objects. It is now clear that the core part of each well-formed NIAM data model is a definitional system. It is also clear that every definitional system makes essential use of Well Founded recursion. In fact, we could say that this is the distinguishing characteristic of a definitional system.

8.4.2 Models of data models

DaMod0, the core model of NIAM conceptual data models, has proved useful. With its help we have shown that the usual notations do what is necessary and can be manipulated in useful ways. It has also helped us to distinguish well-formed data models from ill-formed ones.

DaMod0 is built from roles and entities, nothing else. The definition of DaMod0 requires us to understand the preconditions for adding one Entity Type or Fact Type to an existing data model, nothing more. The extended model, DaMod1, uses only text items such as names and constraint formulas as additional primitives. Constraint formulas are not restricted in any way.

We are aware of only two other models of NIAM data models. (We classify the category treatment described in Section 2.2.6 as an unconvincing experiment). Halpin's model, Section 2.1.5, transforms each data model into a first order theory. The model is inherently incapable of declaring that Entity Types and Fact Types have fixed memberships (a NIAM principle), and it cannot be used to describe illegitimate database instances. It is difficult to see how it would be used to recognise ill-formed data models and to describe preconditions when altering data models.

The Predicator model, Sections 2.1.2 to 2.1.4, models each data model diagram as a graph and then models database instances as functions from the nodes of the graph to sets of entities and tuples. The model does not declare that Entity Types and Fact Types have fixed memberships. An attempt was made to characterise ill-formed data models but the attempt was flawed and also only applicable to finished data models. The model includes a constraint language. Some plausible constraints would be difficult or impossible to express in this language.

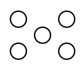
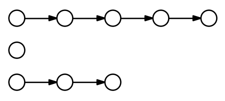
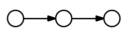
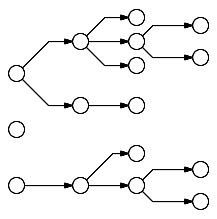
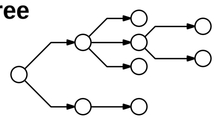
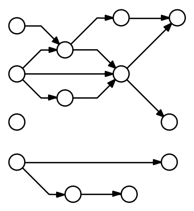
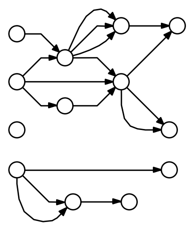
The core of the Predicator model resembles DaMod0 but its semantics are very different. The components are decreed to be notation symbols so the model cannot be used to guide the design of notations in a convincing way. The model uses "predicators" as primitive components, but each predicator has an internal structure that is isomorphic to a data model. They are not really primitive.

Although Halpin's model and the Predicator model have been used to produce useful results we conclude that DaMod0 with its extension DaMod1 is a simpler and more complete model of the NIAM technology.

8.4.3 A common structure

The members of DaMod0 are common structures in the same sense that lists and forests are common. It is possible to fit DaMod0 into a sequence of classes of increasing generality. Figure 8.4.3.1 below lists this sequence in the left-hand column, and provides an example of each class in the middle column. The unfamiliar names Lists and Well Founded Graph will be explained shortly.

Figure 8.4.3.1 A taxonomy

Class name	Example General case	Example Singular case
Set		
Lists		List 
Forest		Tree 
Well Founded Relation		
Well Founded Graph		

Each member *X* of any of these classes can be described as follows. *X* has a set *XPts* of points. Each point is shown as a circle in the pictures. *X* has a set *XArrows* of arrows. Each arrow connects one point to another. *XPts* and *XArrows* can be infinite. There is a subset, *XRoots* $\subseteq_d$ *XPts*, of points that are not the destination of any arrow. There is no path, formed by following arrows from point to point, that does not have a starting point. That is, there are no circuits and no infinite descending chains. As a consequence, *X* has both an induction principle and a recursion theorem. Both principle and theorem use *XRoots* as the base and follow arrows to cover all members of *XPts*. (Of course, if *X* has no arrows then neither principle nor theorem is of any practical use).

The different classes listed in Figure 8.4.3.1 are distinguished by different restrictions on *XArrows*. For Set, *XArrows* is empty; for Lists, the arrows correspond to a partial injection on *XPts*; for Forest, the arrows correspond to an injective relation on *XPts*; for Well Founded Relation there is at most one arrow between any two points; and for Well Founded Graph there are no restrictions.

Clearly, any of these classes can be modelled as a specialised subclass of a class below it in the figure. A different kind of subclass can be formed by requiring that *XRoots* has exactly one member. The classes List and Tree can be formed this way. Examples are shown in the right-hand column of Figure 8.4.3.1. Alternatively, the requirement can be that *XRoots* has at most one member. The subclass then includes a unique null member.

The class Lists (plural) is unconventional but can arise in practice. For instance, the results of some lotteries are published in this form. The winning tickets are listed in ticket number order, but in a separate list for each prize category.

The class which allows many arrows between two points does not appear to have a standard name. The name Well Founded Graph has been used for want of a better one. DaMod0 is a subset of Well Founded Graph, with a particular representation of arrows.

We have shown arrows pointing away from root points. They could have been shown pointing towards root points. The choice is only a matter of emphasis.

8.4.4 Some heuristics

Polya's book (Polya [1990]) gives heuristics for solving mathematical problems. The heuristics are very general and most of the examples are problems in geometry. The development of the models given in Chapters 4 and 6 has suggested some supplementary heuristics specific to the art of designing models of software systems. They are listed below under Polya's primary headings.

8.4.4.1 Understanding the problem

Polya : "First. You have to *understand* the problem."

H1.1 Decide what to model and decide why it is going to be modelled. Know which questions the model is expected to answer.

H1.2 Understand the class of objects that are to be modelled.

H1.3 Distinguish instances from classes. Distinguish notation from the underlying objects it denotes.

8.4.4.2 Devising a plan

Polya : "Second. Find the connection between the data and the unknown." "You should obtain eventually a *plan* of the solution."

H2.1 Perhaps complicated objects can be represented in the model by simple objects, with the complications as secondary features; or perhaps the model can be built in simple stages.

H2.2 Perhaps the establishment of the model is similar to that of another model. Perhaps there are analogous variable features and analogous fixed features.

H2.3 It may look like an *X* (e.g a network), but is it best described as an *X*?

H2.4 People have been doing a job in a certain way for many years. Perhaps there is a good reason for this. But perhaps not.

H2.5 Perhaps the way people build the real objects suggests the way to build the model objects.

8.4.4.3 Carrying out the plan

Polya : "Third. *Carry out* your plan."

H3.1 Ensure that the customer's concepts are present in the model, even if only as secondary features. Consider using the customer's terminology.

H3.2 Are empty sets precluded? Infinite sets? General relations? Partial functions?

H3.3 Are recursively defined functions or predicates used? If so, make sure the relevant structure is one that has a recursion theorem.

8.4.4.4 Looking back

Polya : "Fourth. *Examine* the solution obtained." "Can you *check the result*?"

H4.1 Does the model have all the expected properties? Have they been proved?

H4.2 Does the model have unexpected properties? Are they reasonable? Are they useful?

H4.3 Can the model be simplified? Is there a better model?

8.5 Future work

Finally, there are some topics that deserve further investigation.

Topic 1
A data model with an unused or undefined Entity Type is obviously unfinished. Presumably there are other conditions that are inappropriate in a finished data model. What plausibility checks can be done when a data model is thought to be finished?

Topic 2
Presumably, a finished data model should have a proper identification scheme that enables the contents of the "conceptual" database to be deduced from the contents of the "actual" database. The van Bommel, et al, [1991] paper introduces the predicate *Identifiable* as a means to test this. Unfortunately, its definition is flawed (see Section 2.1.2, problems 5 and 6). What is the proper definition of this predicate? Should data models have additional annotation to indicate the identification scheme?

Topic 3
The ter Hofstede, et al, [1993] paper introduces power type, generalised object type, sequence type, and schema type symbols. It is possible to misuse these symbols (see Section 2.1.3, problem 2). What are the rules for the proper use of these symbols? More generally, what are the principles for extending the notation with symbols that denote derived Entity Types?

Topic 4
Data models often contain constraint formulas defining Subtypes, derived Fact Types, and unclassified constraints. These formulas must be well-formed, must use defined terms, and must be appropriate. How might design tools help check that this is so? Note that checking appropriateness can require theorem proving. There is no general algorithm for doing this but there may be cases where formulas can take a form that facilitates checking.

Topic 5
The tuples defined by a data model are typically held in an encoded form in the database implementation. At some point in the database's development the data model is translated into a database schema which specifies the encoded form of the data. How might the link between data model and database schema be recorded? How might it be used to assist the database users?

Topic 6
We saw in Section 3.2.3, Example 12.3, that there can be a need to define a data model that has an infinite number of Fact Types. There might also be an occasional need to define a Fact Type with infinite arity. How might infinite data models be modelled? What notation would be suitable?

THE END

9 References

Abiteboul, S and Hull, R [1987]. IFO : A formal semantic database model.
ACM Transactions on Database Systems, Vol 12, No 4, p525-565. (December).

Abrial, J R [1974]. Data semantics.
in : Klimbie, J W and Koffeman, K L (Eds) [1974]. Data base management : Proceedings of the IFIP working conference on data base management.
North-Holland Publishing Company, p1-60.

Batini, C, Ceri, S, and Navathe, S B [1992]. Conceptual database design : An Entity-Relationship approach.
Benjamin Cummings.

Brachman, R J, McGuinness, D L, Patel-Schneider, P F, and Resnick, L A [1991]. Living with CLASSIC : When and how to use a KL-ONE-like language.
in : Sowa [1991], p401-456. (*see below*).

Bronts, G H W M, Brouwer, S J, Martens, C L J, and Proper, H A [1995]. A unifying object role modelling theory.
Information Systems, Vol 20, No 3, p213-235.

Chen, P P [1976]. The Entity-Relationship model - Toward a unified view of data.
ACM Transactions on Database Systems, Vol 1, No 1, p9-36.

Codd, E F [1970]. A relational model of data for large shared data banks.
Communications of the ACM, Vol 13, No 6, p377-387, (June 1970).

Date, C J [1995]. An introduction to database systems (6th Ed).
Addison-Wesley Publishing Company.

Eick, C F and Raupp, T [1991]. Towards a formal semantics and inference rules for conceptual data models.
Data and Knowledge Engineering, Vol 6, No 4, p297-317.

Enderton, H B [1972]. A mathematical introduction to logic.
Academic Press Inc.

Enderton, H B [1977]. Elements of set Theory.
Academic Press Inc.

Frederiks, P J M, ter Hofstede, A H M, Lippe E [1997]. A unifying framework for conceptual data modelling concepts.
Information and Software Technology, Vol 39, No 1, p15-25.

Geoffrion, A M [1987]. An introduction to structured modeling.
Management Science, Vol 33, No 5, p547-588. (May).

Geoffrion, A M [1989]. The formal aspects of structured modeling.
Operations Research, Vol 37, No 1, p30-51. (Jan-Feb).

Halpin, T [1991]. A fact oriented approach to schema transformation.
in : Thalheim, B, Demetrovics, J, and Gerhardt, H-D [1991]. MFDBS91 : Proceedings, 1991 symposium on mathematical fundamentals of database and knowledge base systems, p342-356.
Springer-Verlag, Lecture Notes in Computer Science 495.

Halpin, T A [1995]. Conceptual schema & relational database design (2nd Ed).
Prentice Hall of Australia Pty Ltd.

Hamilton, A G [1982]. Numbers, sets and axioms.
Cambridge University Press.

Kent, W [1978]. Data and reality : Basic assumptions in data processing reconsidered.
North-Holland Publishing Company.

Kovács, Gy and van Bommel, P [1997]. From conceptual model to OO database via intermediate specification.
Acta Cybernetica (13), 1997, p103-140.

Kovács, Gy and van Bommel, P [1998]. Conceptual modelling-based design of object-oriented databases.
Information and Software Technology, Vol 40, No 1, 1998, p1-14.

Layzell, P and Loucopoulos, P [1989]. Systems Analysis and Development (3rd Ed).
Chartwell-Bratt.

Leung, C M R and Nijssen, G M [1988]. Relational database design using the NIAM conceptual schema.
Information Systems, Vol 13, No 2, p219-227.

Loucopoulos, P [1993]. Unpublished MSc course notes. (Conceptual modelling - The data perspective).

Mac Lane, S [1971]. Categories for the working mathematician.
Springer-Verlag New York Inc.

Maier, D [1983]. The theory of relational databases.
Pitman Publishing Ltd.

McGinnes, S [1994]. CASE support for collaborative modelling : re-engineering conceptual modelling techniques to exploit the potential of CASE tools.
Software Engineering Journal, Vol 9, No 4, p183-189. (IEE, July).

Misic, V, Velasevic, D, and Lazarevic, B [1992]. Formal specification of a data dictionary for an extended ER data model.
The Computer Journal, Vol 35, No 6, p611-622.

Nijssen, G M and Halpin, T A [1989]. Conceptual schema and relational database design : A fact oriented approach.
Prentice-Hall of Australia Pty Ltd.

Paynter, S [1995]. Structuring the semantic definitions of graphical design notations.
Software Engineering Journal, Vol 10, No 3, p105-115. (IEE, May).

Polya, G [1990]. How to solve it : A new aspect of mathematical method (2nd Ed).
Penguin Books.

Proper, H A [1997]. Data schema design as a schema evolution process.
Data and Knowledge Engineering, Vol 22, No 2, p159-189.

Proper, H A and van der Weide, T P [1995]. A general theory for evolving application models.
IEEE Transactions on Knowledge and Data Engineering, Vol 7, No 6, p984-996, Dec.

Proper, H A and van der Weide, Th P [1994]. EVORM : A conceptual modelling technique for evolving application domains.
Data and Knowledge Engineering, Vol 12, No 3, p313-359.

Russell, B [1919]. Introduction to mathematical philosophy.
Routledge, (republished in 1993).

Scheurer, T [1994]. Foundations of Computing : System development with set theory and logic.
Addison-Wesley.

Scheurer, T [1996]. Unpublished note, 5/2/96, Section 6 : A metatheory. (Levels of variation).

Schubert, L K [1991]. Semantic nets are in the eye of the beholder.
in : Sowa [1991], p95-107. (*see below*).

Simovici, D A and Stefanescu, D C [1989]. Formal semantics for database schemas.
Information Systems, Vol 14, No 1, p65-77.

Sowa, J F (Ed) [1991]. Principles of semantic networks : Explorations in the representation of knowledge.
Morgan Kaufmann Publishers, Inc, San Mateo, California.

ter Hofstede, A H M and van der Weide, T P [1992]. Formalization of techniques : Chopping down the methodology jungle.
Information and Software Technology, Vol 34, No 1, p57-65.

ter Hofstede, A H M and van der Weide, Th, P [1993]. Expressiveness in conceptual data modelling.
Data and Knowledge Engineering, Vol 10, No 1, p65-100.

ter Hofstede, A H M and Verhoef, T F [1996]. Meta-CASE : Is the game worth the candle?.
Information Systems Journal, Vol 6, No 1, p41-68, Jan.

ter Hofstede, A H M, Lippe, E, and Frederiks, P J M [1996]. Conceptual data modelling from a categorical perspective.
The Computer Journal, Vol 39, No 3, p215-231.

ter Hofstede, A H M, Proper, H A, and van der Weide, Th P [1992]. Data modelling in complex application domains.
in : Advanced information systems engineering : 4th international conference CAiSE '92, Manchester, UK, May 12-15, 1992, Proceedings, p364-377.
Springer-Verlag : Lecture Notes in Computer Science 593.

ter Hofstede, A H M, Proper, H A, and van der Weide, Th P [1993]. Formal definition of a conceptual language for the description and manipulation of information models.
Information Systems, Vol 18, No 7, p489-523.

ter Hofstede, A H M, Proper, H A, and van der Weide, Th P [1996]. Query formulation as an information retrieval problem.
The Computer Journal, Vol 39, No 4, p255-274.

Thalheim, B [1993]. Foundations of entity-relationship modeling.
Annals of Mathematics and Artificial Intelligence, Vol 7 (1993), No 1-4, p197-256.

Theodoulidis, C, Wangler, B, and Loucopoulos, P [1992]. The Entity-Relationship-Time model.
In : Loucopoulos, P and Zicari, R (Eds) [1992]. Conceptual modelling, databases, and CASE : An integrated view of information systems development.
John Wiley & Sons, Inc, p87-115.

van Bommel, P, Frederiks, P J M, and van der Weide, Th P [1996]. Object-oriented modelling based on logbooks.
The Computer Journal, Vol 39, No 9, p793-799.

van Bommel, P, Kovács Gy, and Micsik A [1994]. Transformation of database populations and operations from the conceptual to the internal level.
Information Systems, Vol 19, No 2, p175-191.

van Bommel, P, ter Hofstede, A H M, and van der Weide, Th P [1991]. Semantics and verification of object-role models.
Information Systems, Vol 16, No 5, p471-495.

van der Weide, Th P, ter Hofstede, A H M, and van Bommel, P [1992]. Uniquest : Determining the semantics of complex uniqueness constraints.
The Computer Journal, Vol 35, No 2, p148-156.