ELSEVIER Data & Knowledge Engineering 20 (1996) 119–161

DATA & KNOWLEDGE ENGINEERING

A generic model for 3-dimensional conceptual modelling[☆]

P.N. Creasy[a], H.A. Proper[b,*]

[a]*Department of Computer Science, University of Queensland, Australia 4072, Australia*
[b]*Cooperative Information Systems Research Centre, Faculty of Information Technology, Queensland University of Technology, GPO Box 2434, Brisbane, 4001, Australia*

Received 28 July 1995; revised 17 October 1995; accepted 8 December 1995

Abstract

This article discusses two highly intertwined issues. Firstly, we discuss the lack of top-down abstraction mechanisms in data modelling techniques; i.e. abstraction techniques that are fully integrated into the modelling technique and methodology and not just a 'post-modelling process' add on. Secondly, we are concerned with the integration of object-oriented modelling techniques and traditional data modelling techniques.

We start by discussing the pragmatics and motivations behind these issues. Then, a formalisation of (the syntax and semantics of) a data modelling technique is presented that is a generalisation of (E)ER and ORM, and also adheres to the requirements of an object-oriented technique as laid down in the object-oriented manifesto. The result of this exercise is the so-called CDM Kernel. Furthermore, we briefly show how (E)ER, ORM and object-oriented views can be derived from models in the CDM Kernel. This effectively means that the CDM Kernel equates (E)ER, ORM and (some) object-oriented models.

Finally, we briefly discuss some practical issues on the use of the facilities offered by the CDM Kernel in terms of modelling practice and tool support. A generalised conceptual modelling kernel will be very beneficial in the context of CASE Tool and in the context of federated database (information) systems.

Keywords: Conceptual Data Modelling; Generic Data Models; Schema Abstraction; Object Orientation; ORM; ER; NIAM; OMT

1. Introduction

This article is concerned with two central issues. The first issue is the lack of (top-down) abstraction mechanisms in data modelling techniques. The second issue is the integration with object-oriented modelling techniques. We first focus on the necessity of abstraction as an integral part of a data modelling technique.

☆ Part of this work has been supported by an Australian Research Council grant, entitled: "An expert system for improving complex database design".

* Corresponding author. Email: erikp@icis.qut.edu.au, http://www.icis.qut.edu.au/~erikp/

SSDI: 0169-023X(95)00043-7

In process modelling a much used concept is abstraction. Most successful process modelling techniques support some notion of composition which allows modellers to introduce layers of abstraction. Even most traditional programming languages allow for the introduction of layers of abstraction. In data modelling the situation is somewhat different, as most data modelling techniques do not have a built-in facility to introduce abstractions.

Both previous and present research has been done on the automatic generation of abstraction layers for data models [41, 45, 22, 9, 7, 42], and in particular for popular data modelling techniques such as ER [12, 17, 3] and ORM/NIAM [37, 48, 24, 28, 25]. Most of these algorithms try to automatically find so-called *major entity types* in an existing *flat* conceptual schema, and then group relevant relationship types around such a major entity type. This leads to a clustering of the relationship types in a conceptual schema. This clustering can be done repeatedly, resulting in several layers of abstraction.

The above discussed abstraction techniques work well with existing conceptual schemas, and are seen as a way to make existing complex conceptual schemas more presentable to users. This is particularly useful for ORM diagrams; which are notorious for their complexity and detail. Using abstraction techniques, more abstracted views can be created. In addition, it allows one to view an ER schema as an abstracted view of an ORM schema [9, 7].

We would, however, like to use abstraction mechanisms as an integral part of the modelling process, and not as a means to solve the symptom: incomprehensible, large conceptual schema diagrams. In most (large) real life applications, the modelling process is a matter of starting with an overview in terms of the major entity types, and slowly adding more detail. In this paper we propose a syntactic extension to a generalised conceptual data modelling technique that allows us to model these multiple layers of abstraction. When modelling a new application, analysts can specify the complete model using stepwise refinements, while for existing conceptual schemas abstraction techniques can be applied to reverse engineer the abstraction layers from the existing flat conceptual schema. In this article we also show (formally backed) that these extensions allow us to regard an ER schema as a first abstraction from ORM schemas.

The second problem we address in this paper is the closer integration of traditional data modelling techniques with object-oriented data modelling techniques. We extend ORM/ER with object-oriented aspects in such a way that the result adheres to the (modelling technique relevant) requirements on an OODBMS as laid down in the object-oriented database manifesto [2]. The response to the aforementioned object-oriented database manifesto from the more 'traditional' database community is given in [43]. There it is argued that traditional database systems are most likely to be extended with object-oriented aspects. We believe that the same will hold for the modelling techniques used to design such systems, i.e. traditional modelling techniques like ORM and ER will be extended with object-oriented aspects.

Earlier attempts to extend ORM, or ER, based techniques with top down abstraction mechanisms are reported in e.g. [35, 14, 31, 15, 13, 16]. The approaches as presented in [14, 16] also try to establish the connection to the OO paradigm. All of these approaches, however, do not provide a thorough study of the implications of these abstraction mechanisms on the formalisation of the modelling technique as a whole. Nor do they exploit the abstraction mechanism to further integrate different data modelling techniques. In this article we provide an abstraction mechanism which *is* fully integrated into the formalisation of the used data modelling technique being used. Furthermore, the consequences of the introduction of the abstraction mechanism on the modelling technique as a whole are studied in detail; in particular the effects on identification, inheritance, and standard type construction mechanisms like objectification. This will lead to a reduction of the number of elementary concepts in the formalisation. Finally, we also provide a better integration between different data modelling techniques.

The central aim, and main contribution, of this article is therefore to provide a generalised modelling technique for conceptual data modelling. This technique should encompass ER, ORM and object-oriented modelling techniques, while also allowing for the introduction of layers of abstraction. The secondary aims of this article are more concerned with practical issue. We shall show how models in the CDM Kernel can be drawn according to different modelling approaches (ER, ORM and OMT). This means that the key result of this article is a *consolidation* and *formalisation* of existing ideas across different modelling techniques. It does not try to add 'yet another feature', but rather tries to integrate existing ideas.

Please note that it is not the aim of this article to prescribe a set of rules to determine abstractions for a given universe of discourse. Doing so would be in contradiction with what the CDM Kernel tries to do. There are different beliefs on how to use abstractions during the modelling process, and the aim of the CDM Kernel is to allow for different approaches and different data modelling techniques, i.e. to integrate and not to provide *yet another approach*. Nevertheless, a good suggestion on how to (automatically) derive abstraction layers can be found in [10].

For the first aim of this article, we base ourselves on existing research into an ORM Kernel [5, 25]). This research tries to provide a generalised ORM version covering most ORM variations, and to some extent ER versions. This kernel can be tailored to the different ORM versions by adding more specific axioms. For example, requiring all relationship types to be binary leads to the binary variation of ORM. In this article we shall take this existing ORM Kernel and add the extra concepts. However, some existing components from the ORM Kernel are removed to maintain orthogonality of concepts (though there will be no reduction in the conceptual expressiveness). The extra concepts deal with the aforementioned abstraction layers, and the introduction of object-oriented concepts. The result is a Conceptual Data Modelling Kernel (CDM Kernel).

A direct result of the use of the CDM Kernel in an information system development environment, is the observation that in one development project multiple conceptual modelling techniques can be employed simultaneously (possibly in a distributed environment). This could mean that project members can use their own preferred modelling technique, and any investments in, for example, 'old' NIAM/ORM or ER models do not go to waste. Even more, being a generic data modelling technique, the CDM Kernel can very easily be used as a generic data model in the context of open distributed/federated database systems. Finally, most research results based on the ORM Kernel can directly be transplanted to the CDM Kernel, and then be applied to ORM, ER and OO models equally. For example, a conceptual query language like LISA-D [27] can easily be integrated with the new CDM Kernel.

The structure of the paper is as follows. In Section 2 an example application is discussed where top-down abstraction is utilised to divide the problem area into more comprehensible chunks. The syntactical aspects of the CDM Kernel are introduced in Sections 3 and 4. The former limits itself to the information structure only, while the latter discusses the conceptual schema as a whole (including operations). Section 5 deals with semantic issues of the CDM Kernel, i.e. the populations of the conceptual schema. The semantics of operations defined for the types in the conceptual schema is not discussed in this paper. This remains a subject of further research.

2. An example domain

For our example domain, we consider a bank. Fig. 1 shows the top level abstraction of the banking domain. This schema displays five types: Bank, Client, Service, enjoy, of. The Bank type is an abstracted type and forms the top abstraction of the entire banking application. This is also the reason why the enjoy and of relationship types, together with the remaining object types playing a role in these relationship types, are drawn inside the Bank type. Both Client and Service types are abstractions themselves, although their underlying structure is not shown at the moment. When stepping down to a lower level of abstraction, the *void* in these types will be filled with more detail.

The Client and Service type are involved in a relationship type called enjoys. This is a many-to-many relationship where each client must at least enjoy one service and each service offering must be enjoyed by some person. The two black dots indicate that a client of the bank must indeed enjoy some service, and conversely each service must be used by some client. The arrow tipped bar spanning the two roles of the enjoys relationship type indicates that it is a many-to-many relationship. Similarly, the of relationship type models the fact that a bank has many clients, and clients can be client of many banks. The (name suffix to Bank indicates that a bank is identified by a name. Basically, the use of the (name) suffix is a graphical abbreviation of the schema fragment depicted in Fig. 2. The two arrow tipped bars on the roles of the has relationship indicate a 1:1 relationship between a bank and its name; i.e. a

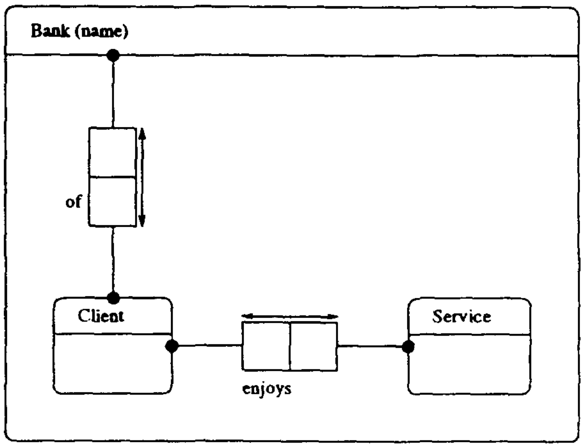


Fig. 1. The top diagram of the Bank domain.

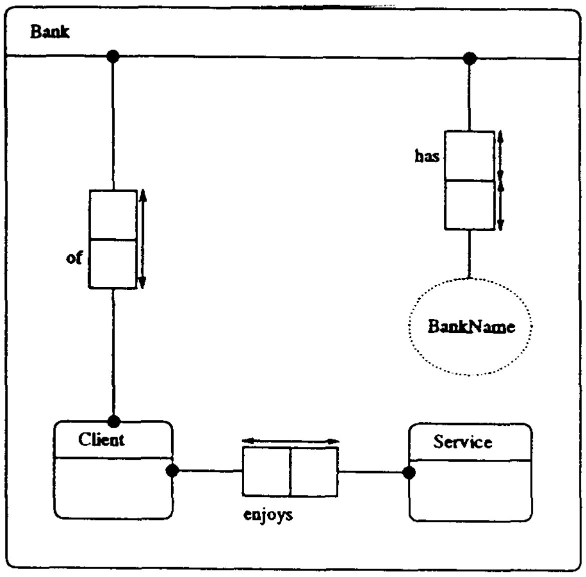


Fig. 2. Fully detailed top diagram.

bank can be identified by its name. The broken ellipse of BankName type indicate that it is a *value type*; i.e. its instances are directly denotable (strings, numbers, audio, video, html).

As a first refinement step we can now take a closer look at what a client is. The details of the Client type are shown in Fig. 3. There, we can see that each client is identified by a Client Nr, as indicated by the (nr) suffix to Client. Each client provides the bank with a unique address as indicated by the arrow tipped bar spanning the role of the lives at relationship type that is attached to Client. This address is mandatory for each client. This "mandatoryness" is indicated by the black dot. Address is a normal object type without any other types clustered to it. Therefore, it is drawn in the traditional ORM way using a solid ellipse. The (description) suffix to Address within the solid ellipse indicates that an address is identified by a description. This corresponds to the same underlying graphical abbreviation.

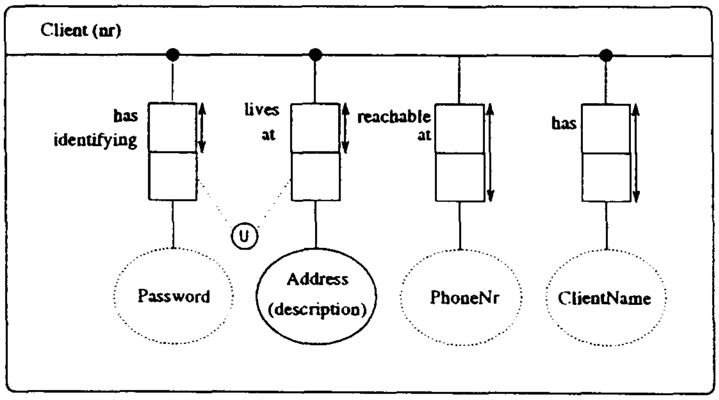


Fig. 3. Refinement of the client type.

Clients must all provide at least one name, but they may have aliases. This leads to the arrow tipped bar spanning both roles of the has fact type and the black dot on the client side. For authorisation of transactions ordered by telephone or fax, the bank and the client agree upon a unique password. The combination of a password and address must uniquely identify a client (indicated in the diagram by the encircled U). Finally, clients may have a number of phone numbers at which they can be reached.

With respect to the abstractions, we can now say that the relationship types has identifying, lives at, reachable at, has (together with the types playing a role in these relationship types) are clustered to Client. For each abstracted type, like Client, such a clustering of types (from a lower level of abstraction) is provided. This could be an emptyset.

In this example we refer to relationship types used in the bank example by means of the text associated with these relationship types, such as has identifying. This text is a so-called *mix fix predicate* verbalisation. These mix fix predicate verbalisations do not have to be unique. The verbalisation has typically occurs numerous times in an average conceptual schema. For example: Client has Client Name and Client has Password. To uniquely identify relationship types (and types in general), each type receives a unique name. For instance Client Naming and Issued Passwords for the two earlier given examples.

The next refinement of the bank domain provides us with more details about the service types available from the bank. This is depicted in Fig. 4. The Service type is a generalisation of three basic types: Credit Card Account, Access Account and Term Deposit Account. The Access Accounts and Credit Card Accounts are first combined into a so-called Statement Account. It should be noted that during a top-down modelling process, a type like Credit Card Account will start out as a 'normal' entity type like Address. However, as soon as other types are clustered to such an entity type, they become abstracted types.

The double lining around the Access Account type indicates that this type occurs in multiple clusterings. A CASE Tool supporting this kind of graphical representation, could

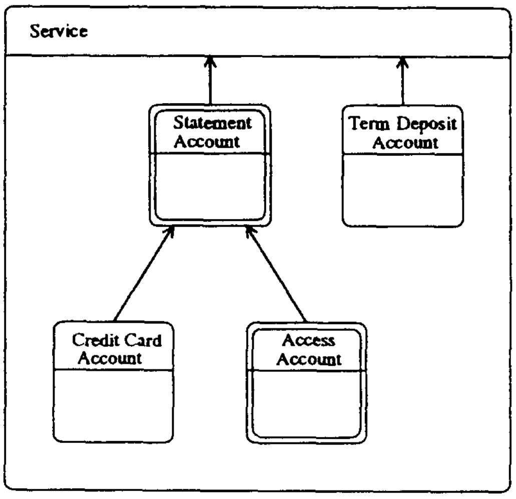


Fig. 4. Refinement of the service type.

have a feature in which clicking on such a double lining results in a list of (abstracted) types in whose clustering this type occurs.

As stated before, a statement account is a generalisation of an access account and a credit card account. The intuition behind a statement account is that for such an account regular statements are sent to the clients and that a transaction record is kept. These details of the statement account are shown in Fig. 5. For each statement account, a number of statements can be issued. A statement lists a number of transactions. This is captured by the lists fact type. This fact type is, however, derivable from the (to be introduced) issue date of a statement and the dates at which the transactions took place. This derivability is indicated by the asterisk.

One of the key features of the CDM Kernel is inheritance of properties between types. Instances (populations) are inherited in the direction of the arrows. For example, each credit card account is a statement account. Other properties, like clustered types, are inherited downwards. Typically, properties at the type level are inherited downward, while properties on the instance level are inherited upwards. The types clustered to Statement Account are therefore formally also part of the clusterings of Credit Card Account and Access Account. Nevertheless, to avoid cluttered diagrams, we have chosen not to show this inheritance explicitly in the diagrams. Therefore, the details of the Credit Card type do not show the details of Statement account. The details of the Credit Card Type are provided in Fig. 6. For each credit card the bank stores its kind, the spending limit, as well as the access account to which the credit card is linked. The suffix ": Statement Account" to "Credit Card Account (nr)" hints at the inheritance of the clustered types to Statement Account. In a CASE Tool supporting our technique, one could implement the facility that clicking on the Statement Account suffix leads to the inclusion of the clustered types introduced by Statement Account. Note that both Access Account and Money Amount have double lining, indicating that they occur in multiple clusters.

For Access Account, the details are shown in Fig. 7. All extra information actually shown there is the identification of an access account; an Access Account Nr as indicated by the (nr)

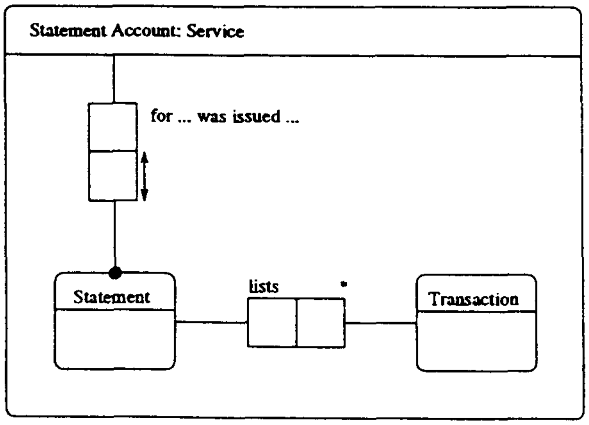


Fig. 5. Refinement of statement account.

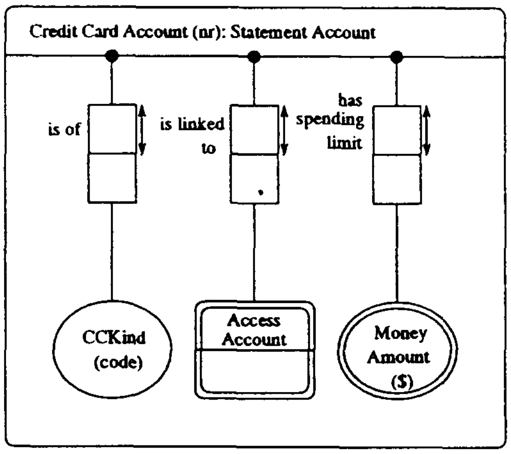


Fig. 6. Refinement of the credit card type.

suffix. Similar to the Credit Card Account, all types clustered to Statement Account are also clustered to Access Account, but we do not display this graphically.

Fig. 8 shows the details of a statement. Each Statement is issued on a unique date. This date, together with the Statement Account for which the Statement was issued, identifies each Statement. Note that we decided to draw some contextual information of the Statement type to show how this type is identified. The for . . . was issued . . . and Statement Account types are not part of the clustering of Statement. The balance as listed on a Statement is, for obvious reasons, derivable from the Transactions that have taken place on this account.

The refined view on a transaction is shown in Fig. 9. A Transaction is identified by the combination of the account it is for and a unique (for that account) transaction number. Note that contrary to a Statement, all components needed for the identification of Transactions are part of the clustering. Each Transaction involves a certain money amount, occurs on a date, and is either a debit or credit transaction (depicted by TR Kind). Furthermore, for each Transaction, some (unique) description may be provided. This example also shows that we must allow for *mutually recursive* abstractions, as the Transaction and Statement Account refinements refer to each other.

Term deposits form a world on their own. This is elaborated in Fig. 10. On each Term Deposit Account, a client can have a series of term deposits. Each time a Term Deposit matures, this term deposit can be rolled-over leading to a new Term Deposit on the current Term Deposit Account. A special kind of Term Deposit is the Long Term Deposit, which is a subtype of Term Deposit. As each subtype inherits all properties from its supertype, the Long Term Deposit type is an abstracted type as well. For these Long Term Deposits we store whether the deposit is to be automatically rolled-over into a new deposit (the short Term

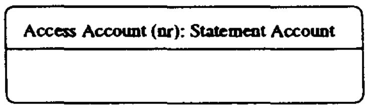


Fig. 7. Refinement of an access account.

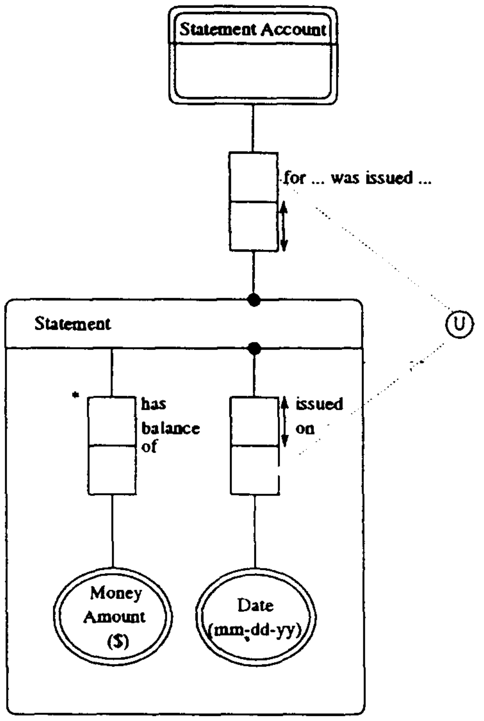


Fig. 8. Refinement of a statement.

Deposits are of this kind by default). In the refinement of a Long Term Deposit, we shall also see what the so-called subtype defining rule for these Long Term Deposits is. Upon maturation, the invested amount including the interest acrued is transferred to a pre-nominated Access Account. Finally, the interest rate given on the deposit is derived from a table listing the Periods for which amounts can be invested. The details of the Period type are given below.

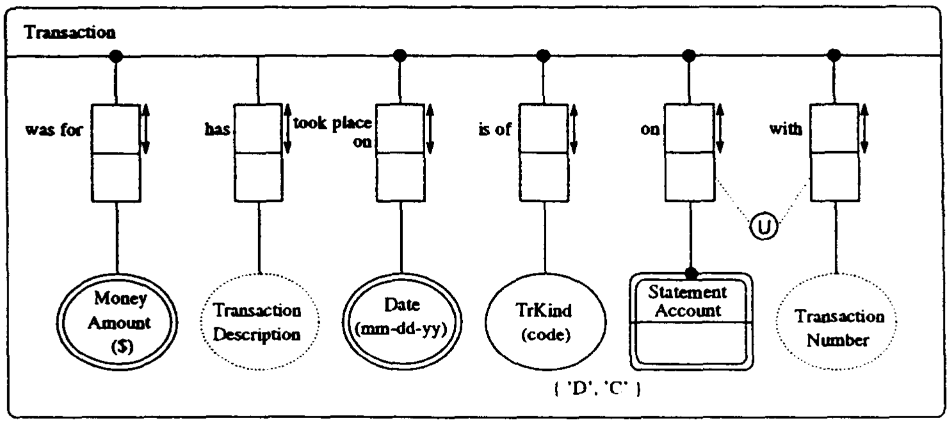


Fig. 9. Refinement of a transaction.

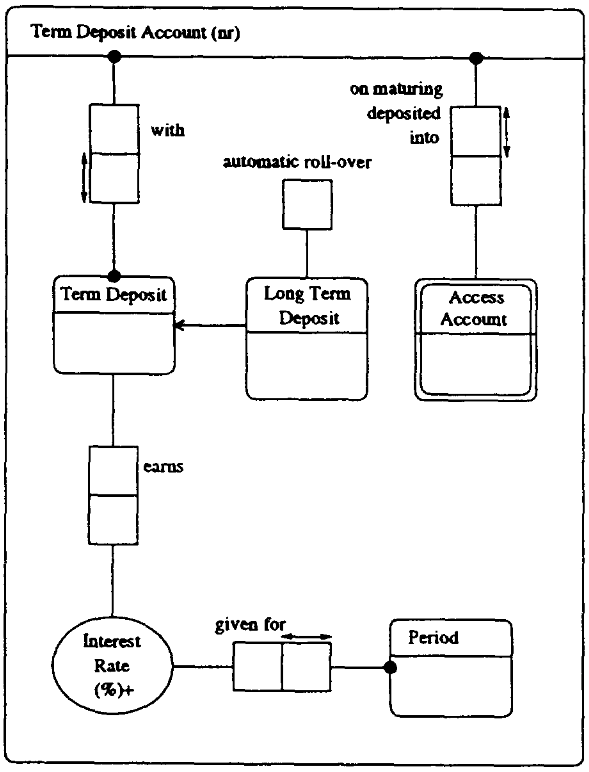


Fig. 10. Refinement of a term deposit account.

A Term Deposit itself is a clustering of the start and ending dates of the deposits and the money amount invested. This is depicted in Fig. 11. A Long Term Deposit is a term deposit with a duration of more than 60 days. In Fig. 12 the details of a long term deposit are shown, including the subtype defining rule. The Long Term Deposit type inherits all clustered types from Term Deposit, while not adding anything to this. Finally, the complete definition of the interest periods are given in Fig. 13.

This completes the schema of the example domain. When modelling a domain like this, the modeller has the choice of using as many layers of abstraction as the modeller sees fit. We only provide a mechanism to introduce these abstractions and are (initially) not so much concerned with the 'sensibility' of abstraction steps. One may, for example, argue that the example given in this section has been split up into too many abstraction levels. As argued in the Introduction, defining guidelines to determine abstractions during a modelling process would be a violation of what the CDM Kernel tries to do: integration. Obviously, in practice one needs to make a choice of what set of modelling guidelines to use. However, the underlying repository (CDM Kernel) can remain the same!

Sometimes, an analyst may want to see the entire schema. This is quite easy to do by uniting all clusterings into one large schema. From the above discussed schema fragments, one can derive the complete ORM schema as depicted in Fig. 14 by uniting all clusters. This is, however, still not the 'lowest' level at which an ORM diagram can be displayed, since we have

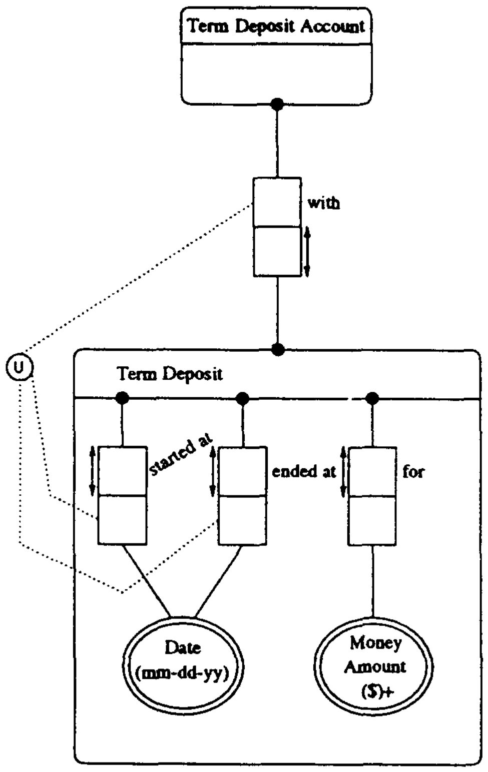


Fig. 11. Refinement of a term deposit.

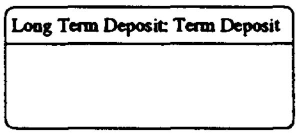


Fig. 12. Refinement of a long term deposit.

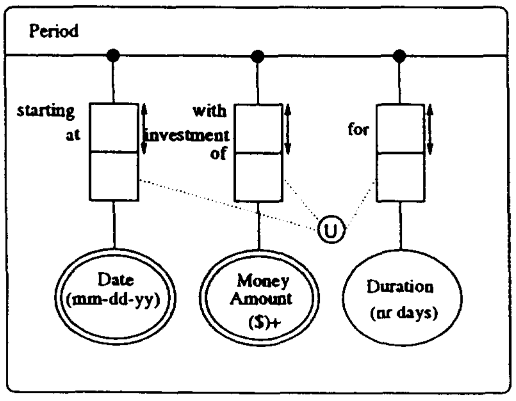


Fig. 13. Refinement of periods.

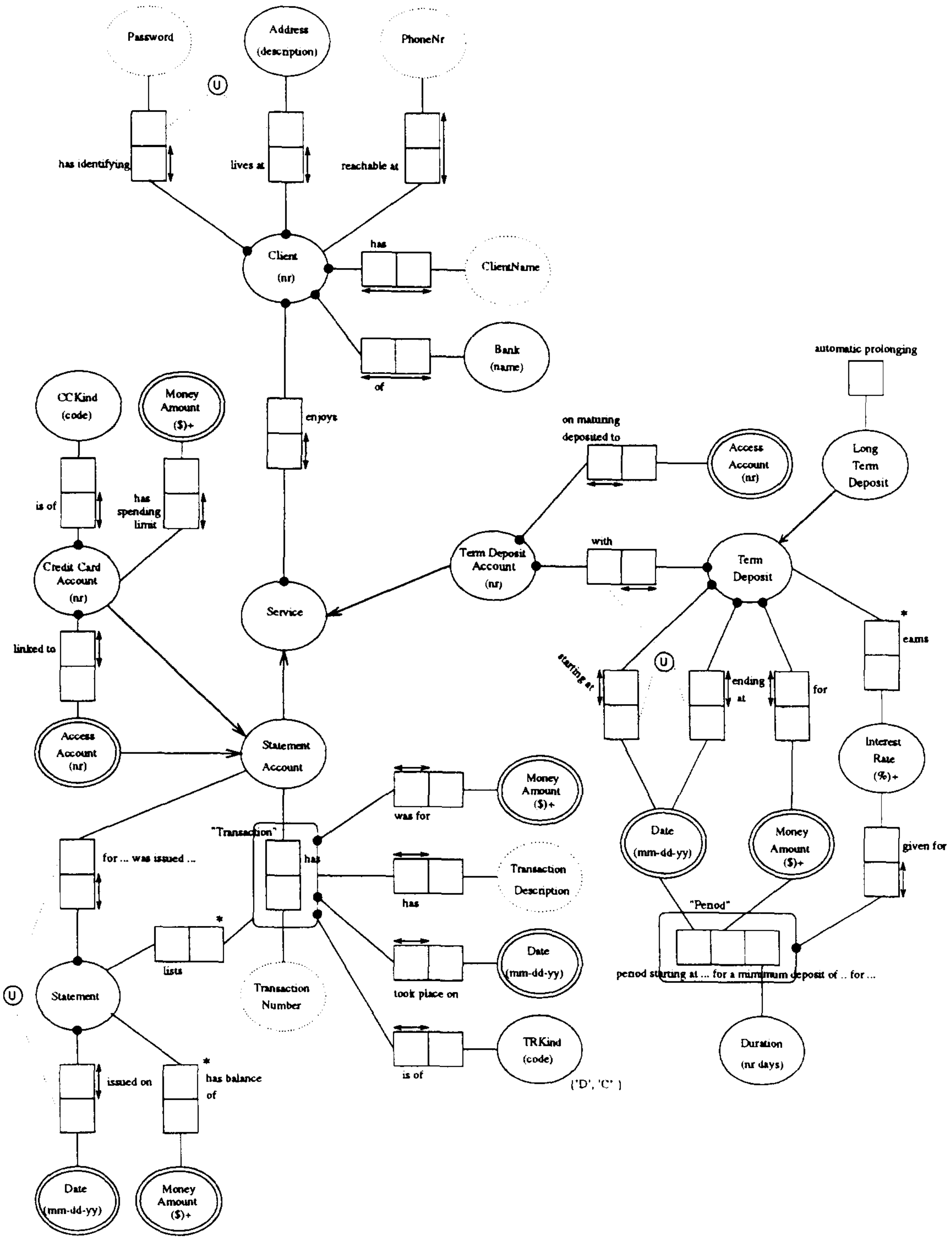


Fig. 14. Complete ORM diagram of the Bank domain.

used the standard abbreviations for simple identifications and the short notation for objectifications. Objectification is a concept we have not yet discussed in this paper.

Both in ORM and ER, modellers can use objectification, also referred to as *nesting*. In a nesting, instances of a relationship type are treated as objects themselves that can play roles in other relationship types. For example, in Fig. 14 Transaction and Period are modelled as nestings. In our view, objectification is an abstraction mechanism in itself. After the addition of the clustering mechanism, as highlighted in this section, objectification can simply be seen as a graphical abbreviation of a (specific flavour of) clustering. Doing so will dramatically simplify the formal definitions underlying ORM. The general pattern of the abbreviation for an objectification is shown in Fig. 15.

In general, when object type x is a compositely identified object type, i.e. more than one value type is used to identify its instances, and all relationship types used to identify instances of x are clustered to x, then we can use the graphical abbreviation. Examples of compositely identified object types are: Transaction, Period, Statement, and Term Deposit. The only object types which have all relationship types needed for their identification clustered to themselves are Period and Transaction. That is why in those cases we can use the graphical abbreviation as used in Fig. 14.

Also when looking at a design procedure for ORM schemas as presented in [24] the decision to model a Transaction, say, as an objectification or a flat entity type is based on considerations of abstraction. When, for the modelling of the relationship types was for, has, took place on, and is of it is preferred to regard a transaction as an abstraction from its underlying relationships to a statement account and transaction number, then the objectified view is preferred to the flat entity view. This directly corresponds to the decision whether these underlying relationships should be clustered to the Transaction object type or not. Later, we shall see that *set types*, *sequence types*, etc. can be treated in a similar way. In [30, 29] it is shown that *set types*, *sequence types* and some other composed types are not fundamental when introducing a special class of constraints which correspond to the set

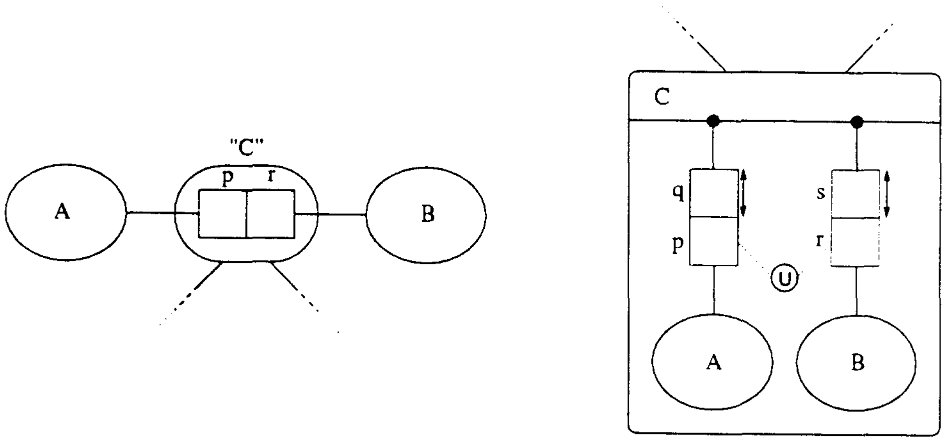


Fig. 15. Objectification as an abbreviation.

theoretic notion of *axiom of extensionality*. This then allows us to regard set typing, sequence typing and schema typing as forms of abstraction.

The schema depicted in Fig. 14 has the same *denotational semantics* as the combination of all previous schema fragments. However, the *conceptual semantics* is different as the abstraction levels (the third dimension) are now missing. Schema abstraction is purely a syntactical issue, and thus carries no denotational semantics. From the point of view of a modeller (and a participant of the universe of discourse), the abstractions do have a conceptual meaning. The abstractions represent certain choices of importance within the universe of discourse.

An (E)ER view can easily be derived as well by uniting all clusterings except for the lowest ones, but interpreting these as attributes. The (E)ER view on this domain is given in Fig. 16. The version we used there is based on the one discussed in [3]. Differing extended ER

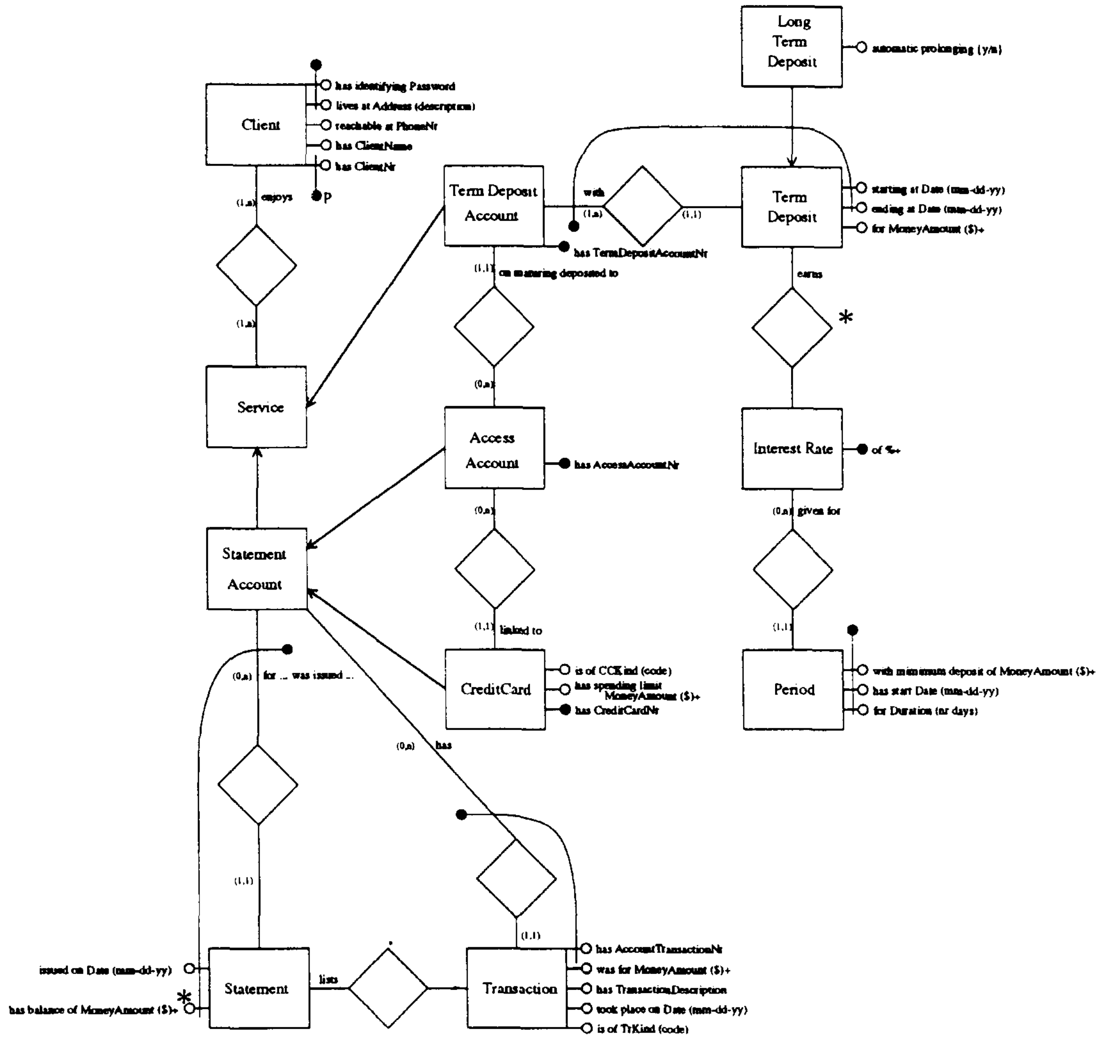


Fig. 16. Complete ER diagram of the Bank domain.

versions use different notations for this concept [19, 18, 21]. The names for attributes in this diagram are simply based on the verbalisations given in the ORM schema. For most ER modellers, the concept of using elaborate verbalisations is new. One could allow for the specification of specific attribute names to, for example, abbreviate with minimum deposit of MoneyAmount ($)+ to MinDeposit. In this article we do not discuss naming conventions in detail but rather focus on the underlying conceptual issues. In [5] we have provided a more detailed study of the relationship between different ER versions and ORM. A detailed case study is also presented there, in which the different concepts underlying these modelling techniques are related, together with a mapping of the (graphical) concepts between the two classes of data modelling techniques.

Of course for the (E)ER case one can also take objectifications into consideration and treat them as a graphical abbreviation. In the example domain, however, they have all disappeared. When there would be a compositely identified object type occurring on an abstraction level higher than one, then this would have led to an objectification in the (E)ER representation. In the remainder we shall also see fragments of an OMT ([39]) view on this same domain. An OMT view does not become very interesting until we have introduced some notion of operations on the types of the schema. Without operations, an OMT diagram can simply be seen as another (E)ER dialect.

Some modelling techniques, for example OMT and some Extended ER versions (e.g. [18]) support forms of aggregation. Although one has to be careful with the notion of *aggregation*, as different authors define this concept differently, most of the definitions known to us are simply forms of abstraction similar to objectification. As an example consider Fig. 17. The left-hand side diagram is an OMT representation of the schema on the right-hand side.

One might now successfully argue that by combining all existing abstraction mechanisms into one single unifying abstraction concept we lose expressiveness with respect to the conceptual semantics. When a modeller decides to model something as an aggregation, as opposed to an objectification, then this decision is of conceptual relevance. Therefore, we shall introduce in the CDM Kernel the possibility to distinguish between different flavours of abstraction. The ones we have seen so far are: objectification, grouping (set types, sequence types, etc.), aggregation and abstraction in general. In the CDM Kernel, we shall not

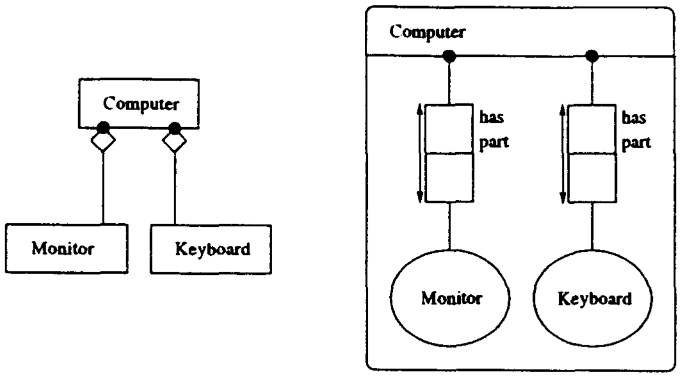


Fig. 17. OMT aggregation as abstraction mechanism.

prescribe a fixed set of abstraction flavours. This is left as a configuration option for a concrete application of the CDM Kernel.

Finally, when entering a model into the CDM Kernel using a normal 'flat' conceptual modelling view, like traditional ORM or (E)ER, the 'third dimension' can be added (reverse engineered) to such a flat model by means of existing algorithms. Examples of such algorithms can be found in [45, 42, 22, 8, 7, 9, 33]. The key difference between each of these algorithms lies in the extent to which they try to incorporate semantic information that is hidden in the constraints and verbalisations provided with the schema. Currently, work is underway with the authors of [9] in formalising their approach more in line with the CDM Kernel presented in this article.

3. Information structures

The formalisation of the CDM Kernel as presented here finds its origins in [4, 28, 27], where it was presented as a formalisation of one of the ORM versions. In [5, 25] the idea of an ORM Kernel was first presented. This ORM Kernel is configurable by means of a number of axioms of taste, adapting the ORM Kernel to the different ORM dialects. Bronts et al. [5] also attempts to include ER into the ORM Kernel, but this was not a complete success as ORM diagrams and ER diagrams model the same domains on different levels of abstraction. In this article we do achieve this goal.

The formalisation of the CDM Kernel therefore inherits a rich and well published history of constant refinements and additions. As we shall show in the next section, the formalisation is now fully able to handle ORM models, ER models and some OO models. Therefore, we now call the formalisation the CDM Kernel, which is short for Conceptual Data Modelling Kernel. With its object-oriented extensions we believe it defines a generic modelling technique that is able to equate models in existing data modelling techniques and extends them with multiple layers of abstractions, i.e. 3-dimensional data modelling.

To support object-oriented models, it must be possible to associate operations (methods) to types. A precise language for the definition of such operations will not be given in this article, neither syntax nor semantics. There is also, as yet, no general consensus as to what such a language should offer, both with respect to the underlying paradigm (functional, imperative, declarative, . . .), nor with respect to the allowed set of operations.

3.1. Type taxonomy

We assume the reader has a basic knowledge of the concepts underlying ORM or ER. A conceptual schema consists of a set of types. Let $\mathcal{TP}$ be the set of all types in a conceptual schema, then this set can be divided in three subclasses. The first class is the set of object types $\mathcal{OB}$. Within this class a subclass of value types $\mathcal{VL} \subseteq \mathcal{OB}$ can be distinguished. Instances of a value type can, as stated before, be directly denoted on a communication medium. These instances usually originate from some underlying domain such as strings, natural numbers, audio, video, html. etc. Later, a function is introduced that assigns a domain (set of values) to each value type.

A separate class of types are the relationship types $\mathcal{RL}$. As we have shown in the previous section, relationship types will (from a formal point of view) not be used in objectifications anymore. Therefore, the set of relationship types is disjoint from the set of object types. Complex types like sequence types, set types, etc. [28], are now seen as ordinary object types. Doing so has led to a considerable reduction in the size of the formalisation of the type hierarchy. This is a change from the approach as previously taken in the ORM Kernel [5]. This change can be made due to the introduction of abstraction levels and the existential uniqueness constraints [30]. The fact that such types, for example, correspond to the traditional notions of a sequence type can be derived from the clusterings.

3.2. Relationships between types

Thus far, we have only discussed a taxonomy of types for the CDM Kernel. Types, however, are interrelated in a number of different ways. In the CDM Kernel, we have the following relationships between types.

3.2.1. Relationship types

Relationship types consist of a number of *roles* (also referred to as *predicators*). These roles are captured in the set $\mathcal{RO}$. The roles in $\mathcal{RO}$ are distributed among the relationship types by the partition: $\mathsf{Roles}: \mathcal{RL} \rightarrow \wp^{+}(\mathcal{RO})$. (Note that $\wp^{+}(\mathcal{RO})$ yields all non-empty subsets of $\mathcal{RO}$). The type playing the role is yielded by the function $\mathsf{Player}: \mathcal{RO} \rightarrow \mathcal{OB}$. In this context it is also useful to introduce a generalisation of the Player function. Based on this function we overload function $\mathsf{Players}$. If P is a set of roles, then: $\mathsf{Players}(P) \triangleq \{\mathsf{Player}(p) \mid p \in P\}$. For any relationship type r we can then define: $\mathsf{Players}(r) \triangleq \mathsf{Players}(\mathsf{Roles}(r))$. Finally, if R is a set of types, then: $\mathsf{Players}(R) \triangleq \bigcup_{r \in R \cap \mathcal{RL}} \mathsf{Players}(r)$. Note that the last generalisation of $\mathsf{Players}$ allows us to apply this function to any set of types; not just relationship types. The reason for this is a gain of elegance in definitions to come.

For the Roles function, we have the following 'inverse' function returning the relationship to which a given role belongs: $\mathsf{Rel}: \mathcal{RO} \rightarrow \mathcal{RL}$, which is defined by: $\mathsf{Rel}(r) = f \Leftrightarrow r \in \mathsf{Roles}(f)$.

3.2.2. Type hierarchies

Both PSM and the ORM Kernel supported two constructs to build type hierarchies. Firstly, specialisation, or subtyping, allows for the introduction of subtypes of more general supertypes. The traditional form of subtyping used in PSM and the ORM Kernel (and (E)ER for that matter), does not allow for the introduction of polymorphic types, e.g. unite a set type and a nested relationship type to become a new polymorph type. This need has led to the introduction of a second construct for building type hierarchies, called polymorphism (also referred to as categorisation in some EER variations). Since we now take the approach that complex types, like objectifications, sequence types, and set types, can be seen as graphical abbreviations on top of simple object types rather than types with a direct underlying structure, we are able to unite the specialisation and polymorphism concepts into one single concept in the CDM Kernel. This concept will, however, still be referred to as subtyping (and supertyping). It is just that the 'rules of the game' are now more liberal.

In a CDM Kernel type hierarchy one can specify hierarchies which will be hard/inefficient to implement directly in a relational system. However, in our opinion it is more important to model the universe of discourse properly in a conceptual language, than to limit a modeller in their 'artistic freedom' by considerations dealing with the underlying relational platform.

The type hierarchy is captured by the $\mathsf{SubOf} \subseteq \mathcal{OB} \times \mathcal{OB}$ relation; with the intuition:

if $x \mathsf{SubOf}\ y$ then 'the population of type x is subset of the population of y'

An example of a small type hierarchy was already shown in the banking example, where we had: Long Term Deposit $\subseteq$ Term Deposit. We shall see that some properties in such a hierarchy are inherited upwards towards the top of the type hierarchy, whereas other properties are inherited downwards. Usually, type-based properties are inherited downwards, whereas population based properties will be inherited upwards. For a more detailed discussion refer to, e.g. [28, 5].

The type hierarchies as used in the CDM Kernel are more general and liberal than the ones used in the original ORM Kernel. The liberality of our current approach is mainly inspired by the category theoretic definitions of the semantics of data modelling techniques as can be found in, e.g. [23, 36]. Please note again that one can always limit the freedom of type hierarchies to comply with less liberal approaches like ORM EER or PSM, by using extra (optional) axioms.

3.2.3. Type clustering

We are now in a position to introduce a mechanism that enables us to cluster types to object types on multiple levels, thus creating multiple levels of abstraction. As can be seen from the running example of the Bank, the abstraction levels were introduced by clustering a set of types to one (major) object type which then becomes an abstracted type. By repeating this on multiple levels we were able to introduce different levels of abstraction at which we described the Bank domain.

Formally, these clusterings are captured by the function:

$$\mathsf{Cluster}: \mathbb{N} \times \mathcal{OB} \rightarrow \mathcal{P}(\mathcal{TP})$$

The intuitive meaning is that if $\mathsf{Cluster}(i, x) = Y$, then at level i type x is considered as an abstraction of the types in Y. For the bank domain we have for example:

Cluster(0, Term Deposit) = {... started at ..., ... ended at ..., ... for ..., Date.mm-dd-yy, Money Amount.$, Date, mm-dd-yy, Money Amount, $}

Cluster(0, Period) = {... starting at ..., ... with investment of ..., ... for ..., Date.mm-dd-yy, Money Amount.$, Duration.nr-days, Date, mm-dd-yy, Money Amount, $, Duration, nr days}

Cluster(1, Statement Account) = {... for ... was issued ..., ... lists ..., Statement, Transaction}

Note: Date.mm-dd-yy represents the implicit relationship type that is present between the Date object type and the mm-dd-yy value type. Usually, this relationship type will be verbalised as ... has ... (see, for example, Fig. 2), but due to our graphical abbreviation for such simple identification schemes these names are usually completely omitted from the diagram.

The function symbol Cluster is overloaded with the additional function Cluster: $\mathbb{N} \to \mathcal{P}(\mathcal{RL})$ which is defined as:

$$\mathsf{Cluster}(i) \triangleq \bigcup_{x \in \mathcal{OB}} \mathsf{Cluster}(i, x)$$

In the next section, well-formedness rules are introduced to which these layers of abstractions should adhere. In the remainder of this section a further refinement of the clustering of types is provided in the form of an encapsulation mechanism as can be found in traditional object-oriented modelling techniques. The encapsulation rules also provide some more natural well-formedness requirements of clustering of relationship types.

3.3. Clustering flavours

In Section 2, we distinguished a number of data modelling constructs that can actually be regarded as forms of abstraction. We also signalled that treating all these flavours of abstraction as one uniform kind of abstraction means that the models will lose conceptual semantics. Therefore, a function is needed that assigns flavours to clusterings. This function is introduced as:

$$\mathsf{Flavour}: \mathcal{OB} \times \mathcal{TP} \rightarrowtail \texttt{Flavours}$$

The interpretation is of course that if Flavour$(x, y) = f$ then the clustering of y to x is of flavour f. Typical examples of flavours are: *objectivity*, *aggregation*, *set*, *sequence*, *abstraction*, etc. As an example, consider Fig. 18. A computer is an aggregation of a monitor, a keyboard, and a cpu. These are clustered to the Computer type at the first level of abstraction. However, at a higher level of abstraction a computer also has properties like speed, memory size, etc. These are then clustered to the Computer type at the second level of abstraction.

3.3.1. Overriding inheritance

Object oriented modelling techniques allow for overriding of inherited properties. At this moment we are not yet interested in inheritance and overriding of operations defined for

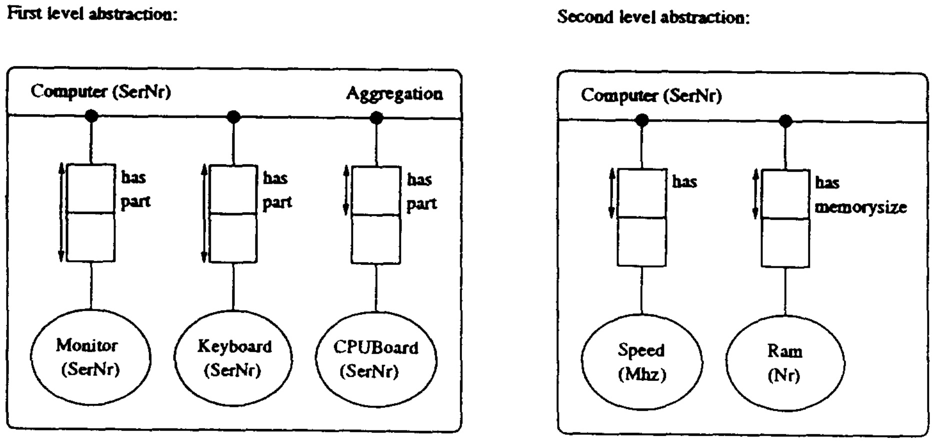


Fig. 18. Aggregation and general abstraction flavours.

operations. This will be discussed in the next section. For the moment we limit our attention to data aspects only. As an example of overriding an inherited property, consider the two type definitions below:

```
class Person
    type tuple (CommutingMeans: Vehicle, Name: String, TaxNr: Integer)
class Student inherit Person
    type tuple (CommutingMeans: Bicycle)
```

These two type definitions are given in a style that can be found as data definition language for OODBMSs. Let us presume that Bicycle is a subtype of Vehicle. Then this is a proper type specification in the sense that each student is a person, while students are only allowed to commute by bike. If we would not have the fact that Bicycle is a subtype of Vehicle, then we could not legally state that Student is a subtype of Person, forbidding the inheritance declaration. For more details see for example [38] or [1].

In terms of an ORM schema, this would lead to the situation depicted in Fig. 19. The commutes by means of relationship type is inherited by the Student clustering. Normally (e.g. in the bank example), we shall not draw inherited relationship types to avoid cluttering up of diagrams. In this case, however, it allows us to illustrate the intended effect of the limitation of the commuting means for students to bicycles only. Therefore, we drew the inherited relationship type using dashed lines. In the supertype, the commuting means role is played by the Vehicle type. For the Student subtype this is limited to the Bicycle type. To capture this formally, we introduce the function RoleLim: $\mathcal{OB} \times \mathcal{RO} \rightarrowtail \mathcal{OB}$. If RoleLim$(x, p) = y$, then in the context of the relationship types clustered to x, the population of Player(p) should be limited to the population of object type y. In the case of RoleLim it does not make sense to take the abstraction level into consideration, as populations are inherited between abstraction levels as a whole.

3.3.2. *Identification schemes*

In traditional modelling techniques like ER and ORM it is required that for each type introduced there is a proper identification schema to identify instances of that type. In operational terms this means that instances of each non-value type must be denotable in terms of some instance of a value type(s). For this purpose, each type has associated a so-called *identification scheme* which provides a scheme for referencing the instances. The underlying rationale is that this allows users to uniquely identify different instances in terms of common (denotable) properties of these instances. Furthermore, when mapping a conceptual type to a set of relational tables, the type can be replaced by the value types used to identify its instances.

Formally, an identification schema is provided by the function: Ident: $\mathcal{OB} \rightarrow (\mathcal{RO} \times \mathcal{RO})^{\diamond}$, where $X^{\diamond}$ denotes the set of partial orders over X (just like X^{+} denotes the set of sequences of X). This function allows us to assign to each type a partially ordered set of identification schemes. This is a partial order since we need to cater for alternative identification schemes (where the order provides the preference). One may argue that in a practical situation this

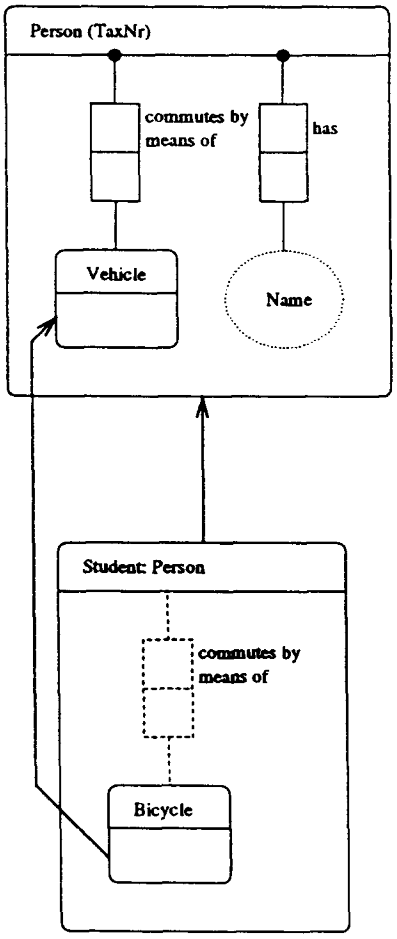


Fig. 19. Commuting means of people and students.

should be a sequence (i.e. a complete order). However, from a theoretical point of view this does not have to be the case.

Each identification scheme itself is a sequence of pairs of roles. Usually, instances of an object type *x* are identified by means of instances of types that are related to type *x* via some relationship(s). For example, in Fig. 14 a term deposit is identified by a term deposit account, a starting date and an ending date. Therefore, the identification scheme is provided as a sequence of role pairs. Each role pair in this sequence provides a path through a relationship type to another object type that is used for the identification.

In typing hierarchies, the identification of object types can be inherited from other object types in the hierarchy. This is discussed in more detail below. There we also introduce well-formedness rules on identification schemes. It is interesting to note that even though value types are obviously self-identifying, there is no *theoretical* objection against identifying instances of value types in terms of other value types. From a pragmatic point of view one might, quite understandably, object to this. The basic CDM Kernel will not provide any extra rules that rule out this behaviour; this is left as optional rules for refinements to 'local' tastes.

Using the Ident function we can define the function Identifiers: $\mathcal{OB} \rightarrow \mathcal{P}(\mathcal{TP})$ returning the set of relationships (if any) that are used to identify a given object type:

$$\mathsf{Identifiers}(x) \overset{\Delta}{=} \cup_{R \in \mathsf{Ident}(x)} \{\mathsf{Rel}(p) \mid \langle p, q \rangle \in R\}$$

Note that we use $e \in S$ to denote that element e is part of sequence S, analogously to sets.

When dropping the requirement that there must be a proper identification scheme (i.e. defining a bijection) for each type in a conceptual schema, it automatically follows that for some types we have to introduce a surrogate identification to distinguish between different instances (possibly with the same properties). Such a surrogate identifier corresponds directly to the notion of an object identifier in object-orientation. For a detailed discussion of the role and pragmatics of object-identifiers in object oriented approaches refer to [32, 1]. Some advanced issues concerning object identity in the context of ORM are addressed in [30, 29].

3.3.3. *The complete structure*

An information structure is fully determined by the components of the following tuple:

$$\mathcal{IS} = \langle \mathcal{RL}, \mathcal{OB}, \mathcal{VL}, \mathcal{RO}, \mathsf{SubOf}, \mathsf{Roles}, \mathsf{Player}, \mathsf{Cluster}, \mathsf{Flavour}, \mathsf{RoleLim}, \mathsf{Ident} \rangle$$

The first 4 components provide the types and roles present in the information structure, and the last 7 components describe their mutual relationships providing the 'fabric' of the information structure.

3.4. *Correctness of information structures*

So far we have discussed the different modelling concepts used in the CDM Kernel. Now we come to the rules governing the CDM Kernel. Some of the rules that are discussed are possible refinements to the CDM Kernel. They are the so-called *electronic SWitches*, provided as the SW axioms. In the discussion of the CDM Kernel so far we have already hinted at possible electronic switches to introduce rules that bring the CDM Kernel closer to concrete modelling techniques. The core rules of the CDM Kernel are captured in the CDM rules.

3.4.1. *Traditional rules*

In the CDM Kernel some of the traditional rules as can be found in the existing formalisations of PSM and the ORM Kernel still hold. Here, we briefly rehearse some of these rules.

The type hierarchy spanned by SubOf should be transitive and irreflexive. Furthermore, type hierarchies should provide a strict separation between value types and non-value types applies. So if $x \mathsf{SubOf} y$, then: $x \in \mathcal{VL} \Leftrightarrow y \in \mathcal{VL}$. For each type, the highest node in the type hierarchy can be identified by:

$$\mathsf{Top}(x, y) \overset{\Delta}{=} x \mathsf{SubOf} y \wedge \neg \exists_z [y \mathsf{SubOf} z]$$

Here, and in the remainder, we use the abbreviation $x \mathsf{SubOf} y \overset{\Delta}{=} x \mathsf{SubOf} y \vee x = y$. In the

CDM Kernel a type hierarchy, each type x must have a unique top. This implies that Top is not just a relation, but is a function Top: $\mathcal{OB} \rightarrow \mathcal{OB}$. Therefore, we shall now write Top(x) whenever we refer to the unique top of the type hierarchy containing x.

In [5] a whole range of additional optional axioms is discussed that can be used in the context of the CDM Kernel as well, and allow it to be adopted to different ORM (ER/OMT) variations.

3.4.2. Type clustering

The clustering of types to other types should adhere to certain rules as well. As a first rule on clustering, a cluster must contain the type to which it is clustered:

[CDM1] $x \in \text{Cluster}(i, x)$

The rationale behind this is purely a pragmatic one; from a formal point of view this turns out to be more convenient.

A clustering must adhere to certain completeness rules. Firstly, a cluster associated to a type may only increase between the abstraction levels:

[CDM2] $i < j \Rightarrow \text{Cluster}(i, x) \subseteq \text{Cluster}(j, x)$

Furthermore, clusterings are inherited between types:

[CDM3] $x \text{SubOf} y \Rightarrow \text{Cluster}(i, y) \subseteq \text{Cluster}(i, x)$

The intuition behind is that the types in a clustering can be regarded as attributes of the type to which they are clustered. In object-oriented approaches, attributes are inherited from types higher in the type hierarchy. This latter rule is provided as an inheritance 'overriding rule' in the OO extensions for ORM proposed in [15]. Note that in drawing the abstracted types on the different levels of abstraction and inheritance, we only draw the types added to the clustering.

All players of relationship types present in a cluster must be present as well:

[CDM4] $\text{Players}(\text{Cluster}(i, x)) \subseteq \text{Cluster}(i, x)$

All types needed to identify any of the object types present in a cluster should be part of the cluster too:

[CDM5] $y \in \text{Cluster}(i, x) - \{x\} \Rightarrow \text{Identifiers}(y) \subseteq \text{Cluster}(i, x)$

Note that the Identifiers of x itself do not have to be present in Cluster(i, x). For example, in Fig. 11 we can see that a term deposit is identified by a combination of the start, end date and term deposit account, while the Term Deposit Account is outside the class of the Term Deposit.

Type hierarchies do not have to be completely present in a cluster. One may, however, want to enforce the rule that the transitivity of the type hierarchy must be visible:

[SW1] $a \text{SubOf} b \text{SubOf} c \wedge a, c \in \text{Cluster}(i, x) \Rightarrow b \in \text{Cluster}(i, x)$

This axiom, together with the inclusion of all relevant identifiers, ensures that we can see the entire sub-hierarchy that is relevant for a cluster. In [15], some examples are shown where this is not required. In some cases of abstraction, one would like to introduce an extra *intermediate* layer in a subtyping hierarchy. Therefore, this rule should not be enforced as a strict law.

Since subtypes may provide their own identification rule, axiom CDM5 does not imply that the top of a type hierarchy is automatically part of a given cluster. One may explicitly want to require presence of the tops:

[SW2] $y \in \mathsf{Cluster}(i, x) \Rightarrow \mathsf{Top}(y) \in \mathsf{Cluster}(i, x)$

The presence of all other types between y and $\mathsf{Top}(y)$ follows automatically from the previous axiom. Again, in [15], examples are shown where this is not required. For instance, in a situation where a lower level abstraction needs to introduce a new common supertype of two existing object types.

Before continuing, we should realise that an information structure can be regarded as a graph where each type results in a node and each connection between types (roles, decomposition, inheritance, etc.) leads to an edge between these two types. A clustering should be complete in the sense that it spans a connected subgraph:

[CDM6] $\mathsf{Cluster}(i, x)$ spans a connected subgraph of the original information structure .

Note: an empty graph is presumed to be a connected graph (not disconnected = connected).

Analogous to some activity/process modelling techniques, we require the existence of a single overall abstracted type. This type represents the entire application, and is called the *top cluster*:

[CDM7] $\exists !_{x,i}[\mathsf{Cluster}(i, x) = \mathcal{TP}]$

Finally, we can formulate two additional optional axioms which further restrain the freedom of abstraction levels. Each time types are clustered, less types will remain. One may argue that the set of remaining types should remain a connected graph:

[SW3] The set of types $(\mathcal{TP} - \mathsf{Cluster}(i)) \cup \{x\}$ spans a connected subgraph of the original conceptual schema at all levels of abstraction i .

Depending on one's view on value types, one may argue that value types are atomic in any sense and can therefore not be the abstraction of anything else:

[SW4] $x \in \mathcal{VL} \Rightarrow \mathsf{Cluster}(i, x) = \{x\}$

We can also define ERs notion of entity type relative to the abstraction levels:

$$\mathcal{EN}_{i+1} \triangleq \{x \in \mathcal{OB} \mid |\mathsf{Cluster}(i, x)| > 1\} - \mathcal{VL}$$

From this we can see that ERs notion of entity types do not come into play until the first abstraction level, which corresponds with the observation that the attributes in ER already provide a first abstraction.

3.4.3. Clustering flavours

Not many rules can be formulated about the clustering flavours. Possibly for more concrete techniques like ORM, OMT or ER, one could add more specific well-formedness rules on the choice of the abstraction flavours. For example, in the case of objectification, the abstracted set of types must be of a pattern as shown in Fig. 15. Nevertheless, one general rule must hold. There are two general rules on abstraction flavours. If a flavour is associated to a clustering of a type to another object type, then there must be a clustering at some level showing this:

[CDM8] $\langle x, y\rangle \in \mathsf{dom}(\mathsf{Flavour}) \Rightarrow \exists_i[y \in \mathsf{Cluster}(i, x)]$

3.4.4. Overriding inheritance

The semantics of the overriding of inheritance with respect to relationship types as can be introduced using the RoleLim function will be introduced in Section 5. Nevertheless, there are certain well-formedness rules that should hold for the overriding of relationship inheritance. Firstly, for any relationship type in a cluster the role limitation function is defined, and a role limitation can only be defined for relationships in this cluster:

[CDM9] $\mathsf{Rel}(p) \in \mathsf{Cluster}(i, x) \Leftrightarrow \langle x, p\rangle \in \mathsf{dom}(\mathsf{RoleLim})$

The default role limitations of a role p would obviously be Player(p) (which basically is no limitation on the population at all). When a further limitation of a role is in place, then this should be a subtype of the default limitation:

[CDM10] $\langle x, p\rangle \in \mathsf{dom}(\mathsf{RoleLim}) \Rightarrow \mathsf{RoleLim}(x, p)\ \mathsf{SubOf}\ \mathsf{Player}(p)$

Finally, additional limitations for types deeper in the type hierarchy cannot be less limiting. So we must have:

[CDM11] If $\langle x, p\rangle, \langle y, p\rangle \in \mathsf{dom}(\mathsf{RoleLim})$, then:

$$x \mathsf{SubOf} y \Rightarrow \mathsf{RoleLim}(x, p)\ \mathsf{SubOf}\ \mathsf{RoleLim}(y, p)$$

In [15] a similar limitation mechanism is introduced called *Type Restriction*. There, a distinction is made between context inheritance and inside inheritance (the *context* of a cluster vs. the *inside* of a cluster) and the effects on type inheritance. In our formalisation they can be treated equally, as we require each schema to have a unique top cluster, so we always deal with type restrictions inside a cluster.

4. Conceptual schemas

The previous section introduced the notion of information structure as used in the CDM Kernel. This section introduces the other aspects of a conceptual schema, including constraints and operations. The introduction of operations (associated with types) is clearly needed to further bridge the gap to object oriented systems.

A conceptual schema over a set of concrete domains $\mathcal{D}$, is identified by the following components:

$$\mathcal{CS} \triangleq \langle \mathcal{IS}, \mathcal{CN}, \mathcal{OP}, \sigma, \mathsf{Ops}, \mathsf{DeRule}, \mathsf{UpdRule}, \mathsf{Encap}, \mathsf{Dom}$$

Each conceptual schema contains an information structure as its backbone. An information structure only focuses on the structure of the information in the universe of discourse.

4.1. Conceptual schema overview

Before we state any detailed rules on conceptual schemas in the CDM Kernel, each of the concepts relevant to the conceptual schema (as, so far, they are not present in the information structure) is briefly discussed.

4.1.1. Constraints

Constraints are modelled as a set $\mathcal{CN} \subseteq \texttt{Predicate}$ where $\texttt{Predicate}$ is some language to define constraints. In some object-oriented approaches, or proposals to extend traditional data modelling techniques with object-oriented aspects, one allows constraints to be associated with types (or classes). In the CDM Kernel a different approach is taken. We shall provide a derived 'allocation' function which automatically determines to which (abstracted) types constraints should be allocated.

4.1.2. Operations

Operations are modelled as a set $\mathcal{OP} \subseteq \mathcal{OB} \rightarrowtail \texttt{Method}$ accompanied by a signature function σ: $\mathcal{OP} \to \mathcal{TP}^{+}$. If $o \in \mathcal{OP}$, we have an operation with signature $\sigma(o)$, while the methods $(o(x))$ may differ for different types in the same inheritance hierarchy. The way in which we use the terms operation and method is borrowed from [39]. The operations themselves are also introduced on different levels of abstraction. This will be tied in with the abstraction levels for the types. These abstraction levels are provided by the function: Ops: $\mathbb{N} \to \wp(\mathcal{OP})$. The Ops function symbol is (similarly to $\mathsf{Cluster}$) overloaded with the function Ops: $\mathbb{N} \times \mathcal{OB} \to \wp(\mathcal{OP})$, which is identified as:

$$\mathsf{Ops}(i, x) \triangleq \{o \in \mathsf{Ops}(i) \mid x \in \mathsf{dom}(o)\}$$

In this article we are not concerned with the naming of concepts. However, one could imagine that a function OpName: $\mathcal{OP} \to \texttt{Names}$ exists which assigns names to functions. When this is not required to be a bijective function, two different operations may bear the same name. This means that operation names can be overloaded.

As an illustration of some operations that are assigned to types, consider Figs. 20 and 21.

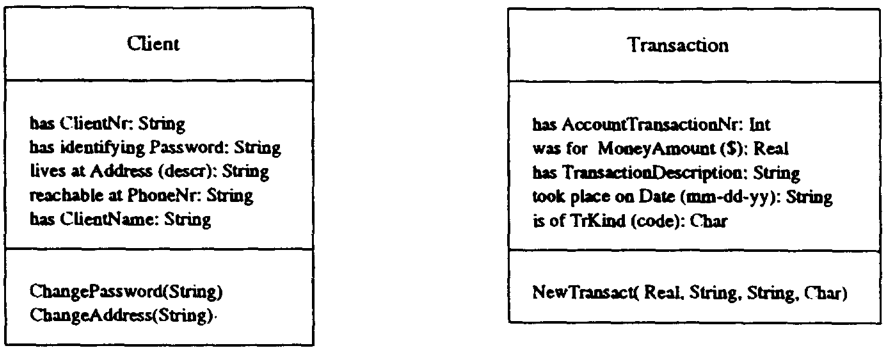

Client
has ClientNr: String
has identifying Password: String
lives at Address (descr): String
reachable at PhoneNr: String
has ClientName: String
ChangePassword(String)
ChangeAddress(String)
Transaction
has AccountTransactionNr: Int
was for MoneyAmount ($): Real
has TransactionDescription: String
took place on Date (mm-dd-yy): String
is of TrKind (code): Char
NewTransact(Real, String, String, Char)

Fig. 20. Operations associated with types.

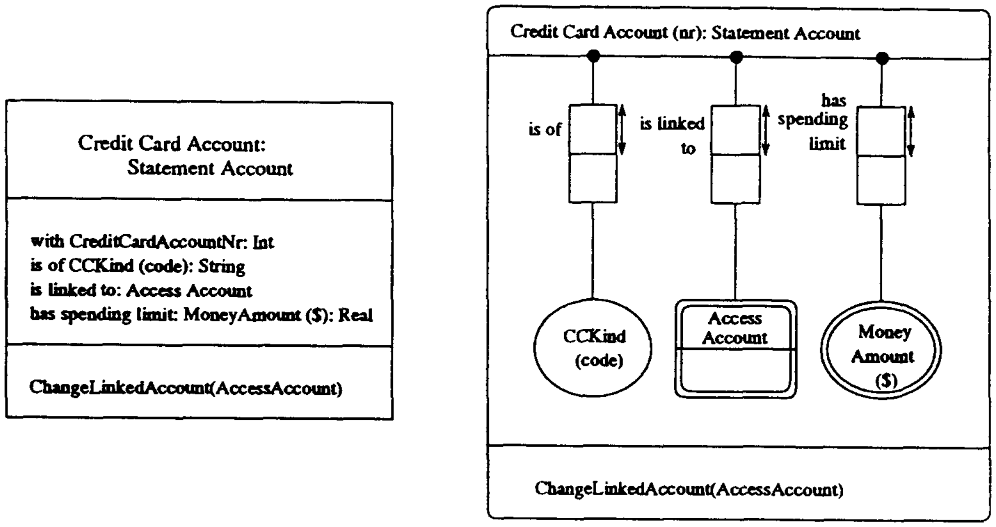

Credit Card Account: Statement Account
with CreditCardAccountNr: Int
is of CCKind (code): String
is linked to: Access Account
has spending limit: MoneyAmount ($): Real
ChangeLinkedAccount(AccessAccount)
Credit Card Account (nr): Statement Account
is of
is linked to
has spending limit
CCKind (code)
Access Account
Money Amount ($)
ChangeLinkedAccount(AccessAccount)

Fig. 21. Higher level operations.

Fig. 20 shows how graphically signatures of operations (and names) can be associated with types. The style used in Fig. 20 is the style of OMT ([39]). The figure shows the types Client and Transaction, together with the types clustered to them, represented as attributes. The domains String, Real, . . .) of the value types as shown in the OMT diagrams, are the domains as determined by the Dom function (to be introduced below).

The Client type has two operations defined for it: ChangePassword(String) and ChangeAddress(String), while the Transaction type only has one associated operation: NewTransact (Real,String,String,Char). These operations are all operations at abstraction level *O*, so they are part of Ops(0) for our example bank domain. In this article we do not provide a language to define the bodies of the operations. Currently work is underway to find a suitable (and generic) candidate for this purpose. As an illustrative example of an operation at a higher level of abstraction consider the one given in Fig. 21. The ChangeLinkedAccount operation can change the access account for a given credit card. The Access Account type itself is an abstracted type, which means that the ChangeLinkedAccount operation cannot be of abstraction level 0. The Credit Card Account type is of level 1, making it plausible to make ChangeLinkedAccount of level 1 as well. Below we shall state well-formedness rules on the abstraction levels at which operations are allowed to occur with regard to the types with which they are associated.

4.1.3. Derivation and update rules

As some of the types in the information structure are derivable from other types, conceptual schemas may contain derivation rules. Conversely, update rules may have been specified for the derivable types. Defining update rules is one way to deal with the so-called view update problem. The derivation rules for the derivable types, and the update rules for some of the derivable types are introduced by:

$$\mathsf{DerRule}: \mathcal{OB} \rightarrowtail \texttt{DerivationRule} \quad \text{and} \quad \mathsf{UpdRule}: \mathcal{OB} \rightarrowtail \texttt{UpdateRule}$$

The set of derivable types is now defined as $\mathcal{DT} \overset{\Delta}{=} \mathsf{dom}(\mathsf{DerRule})$. In this article we do not elaborate on a language for the definition of the `Predicates`, `DerivationRules`, `UpdateRules` and `Methods`. However, we do presume the existence of the following function on these rules:

$$\mathsf{Depends}: (\texttt{Predicate} \cup \texttt{DerivationRule} \cup \texttt{Update} \cup \texttt{Method}) \to \mathcal{P}(\mathcal{TP} \cup \mathcal{OP})$$

returning the types to which the derivation rule (directly) refers.

4.1.4. Encapsulation

Most modelling techniques supporting object-oriented aspects allow for the encapsulation of attributes and operations with their types. In the CDM Kernel encapsulation is provided by the function $\mathsf{Encap}: \mathcal{OB} \rightarrowtail \mathcal{P}(\mathcal{OP} \cup \mathcal{TP})$ which allows us to encapsulate operations and relationship types within the definitions of clusters.

Only relationship types can be encapsulated, since the players of relationship may be shared among relationship types. Similarly, in most object-oriented modelling techniques attributes can be encapsulated, but the underlying domains of these attributes cannot be encapsulated.

4.1.5. Domain assignment

All value types have associated a domain of pre-defined denotable values. For instance, the value type Nat no will typically have associated some implementable subset of the natural numbers. Formally, the possible relationships between the atomic value types and the domains is provided as: $\mathsf{Dom}: (\mathcal{AT} \cap \mathcal{VL}) \to \mathcal{D}$, where $\mathcal{D}$ denotes the set of available base domains.

4.2. Correctness of conceptual schemas

Not every conceptual schema will be a correct one. In this subsection the rules for correct conceptual schemas are introduced.

4.2.1. Operations

Operations can be inherited from one type onto the other. This is why we modelled an operation as a function from a set of types to bodies. The inheritance of operations between types, however, needs to adhere to two strict rules. Firstly, the set of types to which an operation is associated may not be too large in the sense that if an operation is defined for a set of types, then these types must all be part of the same inheritance hierarchy:

[CDM12] If $o \in \mathcal{OP}$, then: $x, y \in \mathsf{dom}(o) \Rightarrow x \mathsf{SubOf} y \vee x = y \vee y \mathsf{SubOf} x$.

The inheritance of operations between types is enforced by:

[CDM13] $x \mathsf{SubOf} y \wedge y \in \mathsf{dom}(o) \Rightarrow x \in \mathsf{dom}(o)$

The underlying intuition is that if we know to do o for ys instances, then we can also do it for xs instances since the population of x is a subset of the population of y.

Similarly to Cluster, operations are inherited from types of lower levels of abstraction.

[CDM14] $i < j \Rightarrow \mathsf{Ops}(i) \subseteq \mathsf{Ops}(j)$

4.2.2. Derivation and update rules

In some data modelling variations each subtype must have some subtype defining rule; i.e. they do not allow for user definable subtypes. For such techniques we would have:

[SW5] $x \mathsf{SubOf} y \Rightarrow x \in \mathsf{dom}(\mathsf{DerRule})$

Derivable types may be used to construct yet even other derivable types. This means that complicated dependencies between the derivation rules may appear. These dependencies should always be carefully checked to avoid circular definitions which can not be solved using fix-point calculations. In [26], a detailed discussion of such interdependencies is given in the context of subtyping. For a more general approach to the problem area of interdependent derivation rules, refer for example to the datalog language ([44, 1]).

Only for types with a derivation rule an update rule may be defined. Therefore we should have:

[CDM15] $\mathsf{dom}(\mathsf{UpdRule}) \subseteq \mathsf{dom}(\mathsf{DerRule})$

4.2.3. Encapsulation

Encapsulation allows us to hide details, like operations and other types, that are clustered to an abstracted type. As such encapsulation forms a very important and integral part of the abstraction mechanisms and inheritance mechanisms provided by the CDM Kernel.

A first well-formedness rule on encapsulation requires that types can encapsulate only those types and operations which are already associated with them, i.e. one can only acquire exclusive access rights for something one already owns:

[CDM16] $y \in \mathsf{Encap}(x) \cap \mathcal{TP} \Rightarrow \exists_i [y \in \mathsf{Cluster}(i, x)]$

[CDM17] $y \in \mathsf{Encap}(x) \cap \mathcal{OP} \Rightarrow \exists_i [o \in \mathsf{Ops}(i, x)]$

Encapsulation of properties is inherited between types. This means that in an inheritance hierarchy, once something is encapsulated it remains encapsulated by a supertype. Formally this is captured as:

[CDM18] $x \mathsf{SubOf} y \Rightarrow \mathsf{Encap}(y) \subseteq \mathsf{Encap}(x)$

The semantics of encapsulation of source is that operation (bodies), constraint definitions, derivation rules, update rules and composed types, can only refer to types and operations that are visible to them. In general, the set of types and operations visible at abstraction level i can be identified by:

$$\mathsf{Visible}(i) \triangleq (\mathsf{Cluster}(i) \cup \mathsf{Ops}(i)) - \bigcup_y \mathsf{Encap}(y)$$

With this we can also define the set of types and operations that are visible from a given type x and abstraction level i:

$$\begin{aligned}\mathsf{Visible}(i,x) \triangleq\ & \mathsf{Visible}(i)\\ & \cup (\mathsf{Cluster}(i,x) \cup \mathsf{Ops}(i,x)) - \bigcup_{y} \mathsf{Encap}(y)\\ & \cup \mathsf{Encap}(x)\end{aligned}$$

Using the Visible function, we can now actually enforce the encapsulation. There are three classes of rules. The first class deals with composed types, the second class with operations and the third class deals with update/derivation rules. The constraints are dealt with separately.

Types that are identified in terms other types can only use types that are visible to them. So we have:

[CDM19] $x \in \mathcal{TP}_i \Rightarrow \mathsf{Identifiers}(x) \in \mathsf{Visible}(i,x)$

For operations we have two rules. These rules require any type or operation referenced in the body of the operation or the signature of the operation to be visible:

[CDM20] If $o \in \mathsf{Ops}(i)$, then: $x \in \mathsf{dom}(o) \Rightarrow \mathsf{Depends}(o(x)) \subseteq \mathsf{Visible}(i,x)$.

[CDM21] If $o \in \mathsf{Ops}(i)$, then: $\mathsf{Set}(\sigma(o)) \subseteq \mathsf{Visible}(i)$.

For derivation and update rules we have the following general rules requiring any referenced type or operation to be visible:

[CDM22] If $x \in \mathsf{dom}(\mathsf{DerRule}) \cap \mathcal{TP}_i$, then: $\mathsf{Depends}(\mathsf{DerRule}(x)) \subseteq \mathsf{Visible}(l,x)$.

[CDM23] If $x \in \mathsf{dom}(\mathsf{UpdRule}) \cap \mathcal{TP}_i$, then: $\mathsf{Depends}(\mathsf{UpdRule}(x)) \subseteq \mathsf{Visible}(l,x)$.

Constraints are not limited by the encapsulation laws. Rather, we define the abstraction level and object type to which a constraint is associated by the function Level: $\mathcal{CN} \rightarrow \mathcal{P}(\mathbb{N} \times \mathcal{OB})$ which is identified by:

$$\mathsf{Level}(c) \triangleq \{\langle l,x\rangle \mid l = \min\{i \mid \mathsf{Depends}(c) \subseteq \mathsf{Visible}(i,x)\}\}$$

A constraint can be associated with more than one type and level combination. Such a level and type can always be found since the CDM Kernel requires the existence of a unique top cluster.

4.2.4. Identification

Thus far, we have only been able to talk informally about the inheritance of, and the requirements of, identification schemes. We are now in a position to make these notions more formal. This is done by introducing the predicate:

$$\mathsf{Identifies} \subseteq \mathcal{P}(\mathcal{RO}^{+}) \times \mathcal{P}(\mathcal{OB}) \times \mathcal{OB}$$

The notation is I $\mathsf{Identifies}_P\, x$, with as interpretation:

> the role sequences I together provide the identification for object type x, when the identification of the types in P is postulated.

The pragmatics of the postulations in P will become clear in the context of recursive identification schemes, as discussed below.

The predicate itself is defined by a set of derivation rules. The first rule deals with the simple case of value types. All value types are, by definition, identifiable through themselves, so we have:

[ID1] $v \in \mathcal{VL} \vdash g \mathsf{Identifies}_g v$

The second rule deals with the situation where an identification scheme is given that directly serves as an identification of an object type.

[ID2] Let $R \subseteq \mathcal{RO} \times \mathcal{RO}$ be a set of role pairs, then:
$\mathsf{Identification}(x, R, P) \vdash \{R\} \mathsf{Identifies}_P x$

The predicate Identification(x, R, P) is used to determine whether a set of role pairs R can indeed be used to identify an object type x, while postulating the identifiability of the object types in P. For each of the role pairs $\langle p, q \rangle \in R$ we refer to $\mathsf{Player}(p)$ as the start type of the role pair, and $\mathsf{Player}(q)$ as the end type. The definition of the $\mathsf{Identification}$ predicate is given in two parts. The first part deals mainly with syntactical issues, whereas the second part covers semantic issues. Firstly, for all role pairs $\langle p, q \rangle \in R$ we should now have:

(1) the role pair should be from the same relationship type:

$\mathsf{Rel}(p) = \mathsf{Rel}(q)$

(2) the pair should start from some type of hierarchy to which x belongs:

$\mathsf{Top}(\mathsf{Player}(p)) = \mathsf{Top}(x)$

(3) the pairs should end in a type that already has a proper identification scheme:

$\exists_{I,P'}[P' \subseteq P \wedge I \mathsf{Identifies}_{P'} \mathsf{Player}(q)]$

Besides these three syntactical requirements, the following semantical requirements should be met:

(1) $\mathcal{CN} \vdash \mathsf{extuniq}(x, \pi_2 R)$
The existential uniqueness constraint requires that a combination of sets of instances from the end types uniquely determines an instance of x. For a detailed discussion of the existential uniqueness constraint refer to [30] or [29].
Note that: $\mathsf{uniq}(x, \pi_2 R) \Rightarrow \mathsf{extuniq}(x, \pi_2 R)$, i.e. traditional uniqueness is a stronger requirement than existential uniqueness.

(2) $\mathcal{CN} \vdash \mathsf{total}(x, \pi_1 R)$
This enforces the requirement that the function f may not be defined for the *null* tuple. The semantics of $\mathsf{total}(x, \tau)$ is defined formally in the next section. Informally, it means that each instance of x must be involved in one of the relationship types of the roles in τ.

Identification is inherited in a type hierarchy. The easiest form of inheritance of identification is downward inheritance. If $x \mathsf{SubOf} y$, then each instance of x is also an instance of y. Therefore, when all instances of y have an identification, all instances of x can be identified too. This leads to:

[ID3] $x \mathsf{SubOf} y \wedge I \mathsf{Identifies}_g y \vdash I \mathsf{Identifies}_g x$

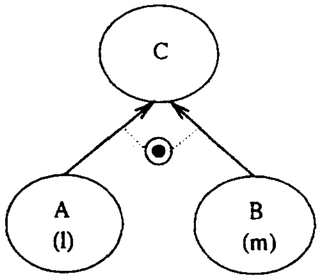


Fig. 22. Upward inheritance in a type hierarchy.

Another class of identification inheritance is exemplified in Fig. 22. The encircled black dot signifies that A and B are a total subtyping of C: $\mathsf{total}(C, \{A, B\})$, which informally means that each instance of C must be an instance of A or B (could be both). Since instances of A are identified by value type l, and instances of B by instances of value type m, and due to the totality of the subtyping of C, all instances of C can be identified; either using value type l or m (or both). This would lead to the following derivation rule:

$$\mathsf{total}(x, S) \wedge \forall_{1 \leqslant i \leqslant n}[I_i \mathsf{Identifies}_g y_i \wedge y_i \in S] \vdash (I_1 \cup \cdots \cup I_n)\mathsf{Identifies}_g x$$

However, this rule is still too limited. For example, consider the schema depicted in Fig. 23. In this schema, X is clearly identified by l. The identifiability of Y and Z is harder. Let us presume that Z would be identifiable. Then we would be able to conclude that Y is identifiable due to upward inheritance. From the identifiability of Y then follows the identifiability of Z via relationship type f by applying the first derivation rule for identification. We can then safely drop the postulated identifiability of Z, and conclude that Y and Z are indeed identifiable. We can do this since each instance of Z and Y can be identified by a list of X instances. Type Y and Z have a so-called *recursive identification scheme*. The recursion in this case terminates since we do not have infinite populations, and X is identifiable.

This now leads to the following two extra derivation rules for identifiers. The ability to deal with recursive identifications is also the reason why the P parameter has been introduced for

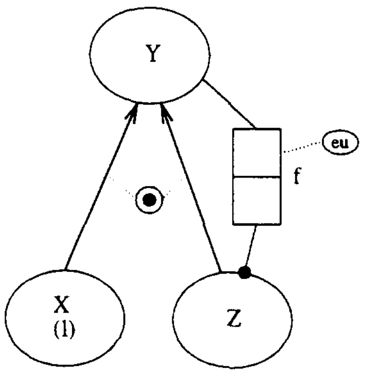


Fig. 23. Identification in a recursive context.

the **Identifies** predicate; it maintains the set of types that have been postulated to be identifiable. The first rule deduces the identifiability of types using postulates.

[ID4] $\mathsf{total}(x, R \cup \{y_1, \ldots, y_n\}) \wedge \forall_{1 \leq i \leq n} \exists_{P' \subseteq P}[I_i \mathsf{Identifies}_{P'} y_i] \wedge R \subseteq P$,
$\vdash (I_1 \cup \cdots \cup I_n)\mathsf{Identifies}_P x$

Note: the n in this rule is at least one! When applying the above rule, one would like P to be as small as possible. This rule actually introduces the postulates we referred to earlier; P must be large enough to contain all needed postulates. For example, in the case of Fig. 23, this rule would be applied where $R = \{Z\}$, $y_1 = X$ with $n = 1$, and postulates $P = \{Z\}$.

The next rule is able to remove any earlier made postulates, as long as certain criteria are met:

[ID5] $\mathsf{total}(x, \{y_1, \ldots, y_n\}) \wedge \forall_{1 \leq i \leq n} \exists_{P' \subseteq P \cup \{y_1, \ldots, y_n\}}[I_i \mathsf{Identifies}_{P'} y_i]$
$\vdash (I_1 \cup \cdots \cup I_n)\mathsf{Identifies}_P x$

The effect of this rule is that all object types y_i which have now received a proper identification are removed from the set of postulates. Naturally, one would like P to be as small as possible. As an example of the interplay between the above two rules, consider the schema shown in Fig. 24. The annotation n: P gives the set of postulates P after the nth step of the derivation process.

Our strategy to use postulates to determine the identification of object types in recursive constructs was inspired by the postulates used by the mode equivalence algorithm used in Algol 68 [34, 47].

The **Ident** function provides the identification scheme(s) for each object type in a conceptual schema. These identification schemes must all be correct:

[CDM24] $\{\mathrm{Set}(I) \mid I \in \mathsf{Ident}(x)\}\mathsf{Identifies}_\emptyset x$

As discussed before, a major difference between object-oriented modelling techniques and traditional modelling techniques lies in the requirement that for all object types there must be an identification scheme. For traditional techniques we would thus have:

[SW6] $\mathsf{dom}(\mathsf{Ident}) = \mathcal{OB}$

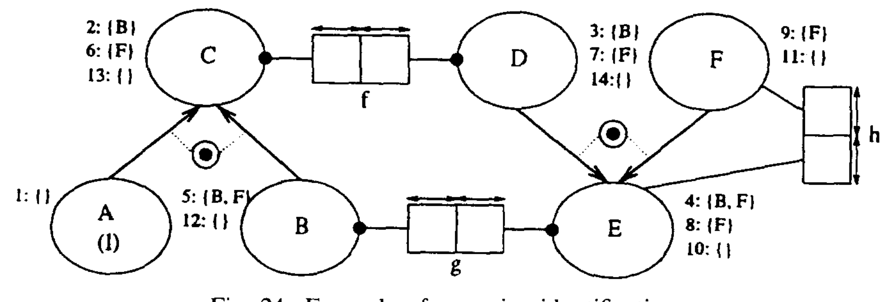


Fig. 24. Example of recursive identification.

4.2.5. *Type relatedness*

Intuitively, object types can, for several reasons, have values in common in some population. This property is initially captured by the concept of type relatedness [28, 27]. Two object types are type related if they may share instances, when only taking structural properties into consideration. Below the notion of type relatedness is further refined to include exclusion constraints that may be added later. Type relatedness, and even more the refinement provided below, is an important concept for query optimisation as it allows for early detection of empty sub-queries.

Type relatedness is defined as a predicate $\sim \subseteq \mathcal{OB} \times \mathcal{OB}$, and is introduced by a set of derivation rules. All object types are type related to themselves.

[TR1] $x \in \mathcal{OB} \vdash x \sim x$

Type relatedness is commutative

[TR2] $x \sim y \vdash y \sim x$

Please note that type relatedness is not a transitive relationship.

Two value types are type related if their underlying domains share values:

[TR3] $x, y \in \mathcal{VL} \wedge \mathsf{Dom}(x) \cap \mathsf{Dom}(y) \neq \varnothing \vdash x \sim y$

Type relatedness is inherited upwards in the type hierarchy:

[TR4] $x \sim y \wedge y \; \mathsf{SubOf} z \vdash x \sim z$

Since type relatedness models the *potential* for object type populations to share values, type relatedness is also inherited downwards in a type hierarchy:

[TR5] $x \sim y \wedge y \mathsf{SubOf} z \vdash x \sim z$

If object types are identified in the same manner, and with the same cardinality, then they must be type related as well:

[TR6] $\exists_{I_1 \in \mathsf{Ident}(x), I_2 \in \mathsf{Ident}(y)}[I_1 \sim I_2] \vdash x \sim y$

where two role sequences, $[\langle p_1, q_1\rangle, \ldots, \langle p_n, q_n\rangle]$, $[\langle r_1, s_1\rangle, \ldots, \langle r_n, s_n\rangle] \in \mathcal{RO} \times \mathcal{RO}$, are considered type related when all the corresponding end types of the role sequences are, and they provide the same identifying cardinality:

$$\begin{aligned}
&[\langle p_1, q_1\rangle, \ldots, \langle p_n, q_n\rangle] \sim [\langle r_1, s_1\rangle, \ldots, \langle r_n, s_n\rangle] \iff \\
&\quad \forall_{1 \leqslant i \leqslant n}[\mathsf{Player}(q_i) \sim \mathsf{Player}(s_i)] \wedge \\
&\quad \mathcal{CN} \vdash \mathsf{unique}(\{q_1, \ldots, q_n\}) \iff \mathcal{CN} \vdash \mathsf{unique}(\{s_1, \ldots, s_n\})
\end{aligned}$$

This rule is in line with the view from traditional modelling techniques that two objects are the same if they have the same set of identifying properties. The first part of the definition requires all end types to be type related, whereas the second part of the definitions requires the identifications to be of the same cardinality, i.e. the identification is either provided as a sequence of *single* instances, or a sequence of *sets* of instances.

Please note that for traditional modelling techniques, when enforcing axiom SW6, that axioms TR4 and TR5 can be proven from axiom TR6. However, since axiom SW6 is an optional axiom, we can not remove these two axioms.

4.2.6. Exclusive types

A more refined view on type relatedness is obtained when taking exclusion constraints into consideration. An example of an exclusion constraint is shown in Fig. 25. From the type relatedness rules follows that A, B and C are mutually type related. However, the exclusion constraint, depicted by an encircled $\times$, requires A, B and C to have exclusive populations. This exclusion constraints in the conceptual schema is formally modelled (as a part of $\mathscr{CN}$) by $\mathsf{exclusion}\ \{A, B, C\}$).

To take the explicitly modelled exclusions into proper consideration, the predicate $\otimes \subseteq \mathscr{OB} \times \mathscr{OB}$ is introduced with the intuition:

if $x \otimes y$ then the populations of x and y are exclusive

Non-type related types are definitely exclusive:

[TO1] $x \not\sim y \vdash x \otimes y$

Exclusion is a commutative relationship:

[TO2] $x \otimes y \vdash y \otimes x$

If a conceptual schema contains an explicit exclusion constraint (or one that can be derived from the other constraints), then this leads to exclusive types:

[TO3] $(\mathscr{CN} \vdash \mathsf{exclusion}(X)) \wedge x, y \in X \vdash x \otimes y$

Exclusion is inherited downward in a type hierarchy:

[TO4] $x \mathsf{SubOf} y \wedge y \otimes z \vdash x \otimes z$

When all direct subtypes of certain types are mutually exclusive, then the supertypes must be exclusive as well:

[TO5] $\forall_{a \mathsf{SubOf}_1 x, b \mathsf{SubOf}_1 y} [a \otimes b] \vdash x \otimes y$

If object types used for the identification are found to be exclusive, then the identified object types are exclusive as well (although they could still be type related):

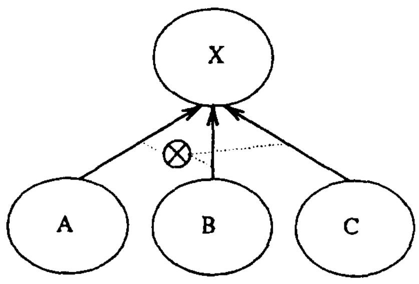


Fig. 25. An example of an exclusions constraint.

[TO6] $\forall_{i_1 \in \mathsf{Ident}(x), i_2 \in \mathsf{Ident}(y)}[i_1 \otimes i_2] \vdash x \otimes y$

where we have for $[\langle p_1, q_1\rangle, \ldots, \langle p_n, q_n\rangle], [\langle r_1, s_1\rangle, \ldots, \langle r_n, s_n\rangle] \in \mathcal{RO} \times \mathcal{RO}$:

$$[\langle p_1, q_1\rangle, \ldots, \langle p_n, q_n\rangle] \otimes [\langle r_1, s_1\rangle, \ldots, \langle r_n, s_n\rangle] \quad \Leftrightarrow \quad \exists_{1 \le i \le n}[\mathsf{Player}(q_i) \otimes \mathsf{Player}(s_i)]$$

In the next section the implication of the $\otimes$ relationship on the semantics of information models is provided.

5. Semantic issues

Thus far, we have only discussed syntactic issues of the CDM Kernel. In this section we briefly discuss the semantic issues. This can be done briefly, as the semantics of the CDM Kernel does not differ much from the semantics of the ORM Kernel, or even of that of the original PSM/PM languages.

The population of a schema is provided by the function Pop which assigns to each type $\mathcal{TP}$ a set of values. This function can, however, be split up in two separate functions based on the two major type classes:

$$\mathsf{Pop}_{obj}: \mathcal{OB} \to \mathcal{P}(\Omega)$$
$$\mathsf{Pop}_{rel}: \mathcal{RL} \to \mathcal{P}(\mathcal{RO} \rightarrowtail \Omega)$$

The set $\Omega \triangleq (\bigcup \mathcal{D}) \cup \mathcal{OID}$ is a set of atomic instances built from the following two (exclusive) subsets:

(1) The set of values from the domains for value types: $\bigcup \mathcal{D}$

(2) A set of object identifiers $\mathcal{O}_{\mathcal{J}}\mathcal{D}$, each representing an instance of a non-value type.

This is the key difference between the formalisation of the population concept in the CDM Kernel and its predecessors. In previous definitions, Ω required a recursive definition to deal with complex types like objectification. Due to our treatment of objectification as an abstraction, we only have to deal with flat conceptual schemas from a population point of view. This allows us to simplify the definition of population. In practice we shall write Pop, as it will be clear from the context which function is actually meant.

It should be noted that, as discussed before, in the context of a non-object-oriented modelling approach an identification scheme is provided for all object types. This property allows us to replace the object identifiers of instances of these non value types by their proper identifications when mapping a conceptual schema, and associated population, to a relational database and database population. For those object types which have not received a proper identification scheme, the object identifiers must be stored explicitly. When implementing a conceptual schema in terms of an object-oriented database, these object identifiers obviously directly map to the object identifiers in the database management system.

5.1. General population rules

For populations there are certain well-formedness rules as well. These rules differ from earlier formalisations as we now have, from a population point of view, a much simpler setup. In this section we focus on the key differences.

Each value type population should be drawn from the proper domain:

[P1] $v \in \mathcal{VL} \Rightarrow \mathsf{Pop}(v) \subseteq \mathsf{Dom}(v)$

The population of a non-value type must be a set of object identifiers:

[P2] $x \in \mathcal{OB} - \mathcal{VL} \Rightarrow \mathsf{Pop}(x) \subseteq \mathcal{OID}$

Please note that Pop is only a *conceptual* population. As stated before, when mapping to a database system these object identifiers may disappear.

Besides these two CDM Kernel specific rules, general rules from PSM and the ORM Kernel on population hold. As usual, instances of relationship types should be functions with as domain the set of roles involved in that relationship type. The instances playing a role in such a relationship should be from the proper types. Populations should honour the typing hierarchies, so the population of a subtype should be a subset of the population of a supertype. The exclusion relationship on types leads to the (strong typing) requirement that if $x \otimes y$, then the populations should indeed be exclusive: $\mathsf{Pop}(x) \cap \mathsf{Pop}(y) = \varnothing$.

5.2 Overriding of inheritance

An important concept, from a semantic point of view, is the limitations of role populations when introducing subtypes. An example of such a situation is displayed in Fig. 19. In this subsection we are therefore concerned with the semantics in terms of populations of the RoleLim construct.

When $\mathsf{RoleLim}(x, p) = a$, intuitively the population of $\mathsf{Player}(p)$ should be limited to the population of a with respect to cluster x. For this purpose, we first need to develop the notion of a cluster population.

As mentioned before, in the context of the RoleLim function the partition of clusterings to the same object type over multiple levels of abstraction is ignored. For $x \in \mathcal{OB}$ we therefore further overload Cluster with the function Cluster: $\mathcal{OB} \rightarrow \wp(\mathcal{TP})$ with definition:

$$\mathsf{Cluster}(x) \overset{\Delta}{=} \bigcup_{i \in \mathbb{N}} \mathsf{Cluster}(i, x)$$

The population of an object type $y \in \mathsf{Cluster}(x) \cap \mathcal{OB}$ is provided by the function $\mathsf{CPop}(x, y)$. This CPop: $\mathcal{OB} \rightarrow \wp(\mathcal{OID})$ function is exactly defined by the three following derivation rules:

[CP1] $i \in \mathsf{Pop}(x) \vdash i \in \mathsf{CPop}(x, x)$

[CP2] If $a \mathsf{SubOf} b \wedge a, b \in \mathsf{Cluster}(x)$, then: $i \in \mathsf{CPop}(x, a) \vdash i \in \mathsf{CPop}(x, b)$

[CP3] If $a \mathsf{SubOf} b \wedge a, b \in \mathsf{Cluster}(x)$, then: $i \in \mathsf{CPop}(x, b) \cap \mathsf{Pop}(a) \vdash i \in \mathsf{CPop}(x, a)$

[CP4] If $\mathsf{Rel}(p) = \mathsf{Rel}(q) \wedge \mathsf{Rel}(p) \in \mathsf{Cluster}(x)$, then:

$$i \in \mathsf{Pop}(\mathsf{Rel}(p)) \wedge i(p) \in \mathsf{CPop}(x, \mathsf{Player}(p)) \vdash i(q) \in \mathsf{CPop}(x, \mathsf{Player}(q))$$

Intuitively, when regarding a population as a graph with each node labelled with the object type of which it is an instance, the above definition corresponds to the construction of a subgraph starting from nodes labelled with x only using neighbouring nodes that are labelled with object types in $\mathsf{Cluster}(x)$.

The semantics of RoleLim is now expressed as:

[P3] $\text{RoleLim}(x, p) = a \Rightarrow \text{CPop}(x, \text{Player}(p)) \subseteq \text{Pop}(a)$

5.3. *Constraints and derivation rules*

The graphical constraints that can be defined on models in the CDM kernel are the same as the graphical constraints that can be defined on ORM models, which is a superset of the ones that can be defined on (E)ER models. The semantics of graphical constraints is not discussed in full detail in this article. For a detailed discussion of these semantics, please refer to [4, 46].

Nevertheless, in the context of our object-oriented extensions it makes sense to add a few new classes of graphical constraints. In the context of subtyping, and the role limitations provided by RoleLim, it makes sense to also provide a total role (the black dot) constraint that is limited to subtypes. The traditional format of $\text{total}(\{p_1, \ldots, p_n\})$ would have to change to $\text{total}(x, \{p_1, \ldots, p_n\})$, where x is type related to all players of the roles $p_1, \ldots, p_n$. The semantics are:

$$\text{Pop}(x) = \bigcup_{1 \leq i \leq n} \text{Pop}(\text{Player}(p_i))$$

in other words, all instances of x must play at least one of the listed roles: $p_1, \ldots, p_n$. Similarly, uniqueness constraints could be more exactly specified for subtypes using the format: $\text{unique}(\{\langle p_1, x_1\rangle, \ldots, \langle p_n, x_n\rangle\})$ or $\text{extuniq}(\{\langle p_1, x_1\rangle, \ldots, \langle p_n, x_n\rangle\})$. A syntactic requirement would be that for each i we must have: $\text{Player}(p_i) \sim x_i$. The semantics in this case would be:

The combination of the instances of roles $p_1, \ldots, p_n$ must be (existentially) unique when limited to the instances of $x_1, \ldots, x_n$, respectively.

A language for derivation rules, or non-graphical constraints, is not discussed in this paper. We propose to develop a slightly modified version of the LISA-D language as introduced in [27]. The modifications are required to deal with the more liberal approach to object identification and object equality in object-oriented systems.

These special classes of constraints are particularly useful for constraint restriction as discussed in, e.g. [15]. Using one of the above constraints, we can easily define more stringent constraints on subtypes. For example, stating that all Access Accounts have a transaction associated (while leaving this optional for Credit Cards can be done quite easily by stating total Access Account, has Transaction).

6. Relation to object-oriented modelling techniques

We believe that with the inheritance of properties (both clustered types and operations) between types in an inheritance hierarchy or different levels of abstraction, we have incorporated one of the key features of object oriented modelling techniques into the CDM Kernel. The axioms we have formulated on the inheritance of properties are completely in line with object oriented typing systems [11, 6, 38, 1].

When comparing the CDM Kernel to the requirements on object oriented database systems as laid down in the Object Oriented Database Manifesto [2], the CDM Kernel can successfully claim to be object oriented when limiting ourselves to those requirements that are relevant for modelling techniques:

- *Thou shalt support complex objects*
  It is important to realise that we consider the construction of complex objects to be a form of abstraction. The existential uniqueness constraint provides the semantic power to build complex objects like sets, sequences, bags, etc. while the abstraction mechanism allows us to represent these complex types in a clear and concise way.
- *Thou shalt support object identity*
  There is nothing in the kernel that disallows this. We have presented the traditional approach to object identity, i.e. identifying objects by means of a set of identifying properties (relationship types), as an optional feature.
- *Thou shalt encapsulate thine objects*
  This is supported by the Endcap predicate. It should be noted, however, that our notion of encapsulation is more *type oriented* than *object oriented.* Two instances of different types cannot 'see' each other's encapsulated attributes and operations, while two different instances of the *same* type can still see each other's encapsulated properties. This latter would not be the case in a 'truly' object oriented approach.

  Our notion of encapsulation can quite easily by complemented with a more restricted version IEncap. The semantics can be expressed in a similar way as we expressed the RoleLim. A Cluster population should be derived, upon which the encapsulation rules can be applied. This latter mechanism is related to the notion of *local referential integrity* as introduced in [31].

  It is interesting to note that in a recent publication [20] it was argued that encapsulation should not be used for object oriented analyses, and should only be used for object oriented design. Our type oriented encapsulation is more aimed at supporting re-use and partitioning of domains, by allowing information hiding at the type level.
- *Thou shalt support types or classes*
  A central role in the CDM Kernel is played by object types.
- *Thine classes or types shalt inherit from their ancestors*
  The SubOf is our type hierarchy, and inheritance of properties has been discussed in detail.
- *Thou shalt not bind prematurely*
  The OO Manifesto uses this rule to require object oriented systems to support overriding, overloading and late binding. In this paper we did not discuss a language for process modelling, so this is only of partial importance to us. However, we have identified the notion of further restricting the population of roles when subtyping, which can be seen as a form of overriding.

  Overloading is highly intertwined with naming conventions of concepts. In this article we did not elaborate on naming functions of object types and relationship types. However, we did hint at possibilities of using these naming functions to overload names. A typical example of such a name is the role name has, which is one of the most commonly used (English language) role names in ORM schemas.

The remaining mandatory requirements are, in our opinion, more relevant in the context of query languages and actual database management systems. While the CDM Kernel defines an information modelling technique that is an object-oriented modelling technique (or very close to it), it does not ignore its ties to traditional modelling techniques.

Using so-called *electronic switches* the kernel can be adopted to the 'limited' views of any of the more commonly used modelling techniques. Besides defining a powerful modelling technique in itself, the CDM Kernel can therefore be used to equate models in different information modelling techniques.

7. Conclusions and further research

In this article we have presented the thorough and detailed formalisation of a generic information modelling technique. We have focussed on the underlying way of modelling while distancing ourselves from discussions about notational and design procedural issues. Our generic technique also provides a powerful enhancement of the conceptual semantics of models in the form of a multilevel abstraction mechanism. Due to this abstraction mechanism, we have been able to reduce the number of basic type classes, by acknowledging the fact that, for example, objectification, set types, and sequence types, are essentially forms of abstraction. We therefore also support different flavours of abstraction.

Finally, we can mention a number of areas in which the CDM Kernel can be applied and avenues for further research:

(1) Development of a CASE-Tool that uses the CDM Kernel as a generic meta-model for the underlying repository. On top of this repository specialised CASE-Tools can be defined tailored for (E)ER, ORM and object-oriented views of the same models. Currently, this is being realised.

(2) The modification of the existing LISA-D language to cope with the more liberal notion of object identification

(3) Development of a suitable activity/method modelling technique to complement the data modelling aspects discussed in this article.

(4) Application of the CDM Kernel as a generic data modelling language ([40]) in the context of federated/distributed information systems. The CDM Kernel can be used as a model of the information stored by a *type manager* and an *object broker*.

(5) Extension of the existing conceptual schema design procedures with guidelines for top down and bottom up abstractions. In [10] work is reported on an algorithm to automatically determine bottom up abstractions for a given flat conceptual schema.

An extended version of this article can be obtained from the authors, or alternatively be accessed from http://www.icis.qut.edu.au/~erikp/publist.html.

Acknowledgements

We would like to thank the anonymous referees for their valuable comments that have led to improvements of the first draft of this article. We would also like to thank N.P. Sharma. He has lent a willing ear for some of the early ideas for this paper. The late afternoon coffee sessions at the bookshop were quite fruitful.

References

[1] S. Abiteboul, R. Hull and V. Vianu, *Foundations of Databases* (Addison-Wesley, Reading, Massachusetts, 1995).

[2] M. Atkinson, D. DeWitt, D. Maier, F. Bancilhon, K. Dittrich and S.B. Zdonik, The object-oriented database system manifesto, in: W. Kim, J.-M. Nicolas and S. Nishio, eds. *Proc. First Int. Conf. on Deductive and Object-Oriented Databases (DOOD-89)*, Kyoto, Japan (Elsevier Science Publishers, 1989) 40–57.

[3] C. Batini, S. Ceri and S.B. Navathe, *Conceptual Database Design – An Entity-Relationship Approach* (Benjamin Cummings, Redwood City, California, 1992).

[4] P. van Bommel, A.H.M. ter Hofstede and Th.P. van der Weide, Semantics and verification of object-role models, *Information Systems* 16(5) (October 1991) 471–495.

[5] G.H.W.M. Bronts, S.J. Brouwer, C.L.J. Martens and H.A. Proper, A unifying object role modelling approach. *Information Systems* 20(3) (1995) 213–235.

[6] K.B. Bruce and P. Wegner, An algebraic model of subtype and inheritance, in: F. Bancilhon and P. Buneman, eds. *Advances in Database Programming Languages*, ACM Press, Frontier Series (Addison-Wesley, Reading, Massachusetts, 1990) 75–96.

[7] L.J. Campbell, Adding a new dimension to flat conceptual modelling, in: T.A. Halpin and R. Meersman, eds., *Proc. First Int. Conf. on Object-Role Modelling (ORM-1)*, 294–309, Magnetic Island, Australia, July 1994.

[8] L.J. Campbell and T.A. Halpin, Automated support for conceptual to external mapping, in: S. Brinkkemper and F. Harmsen, eds., *Proc. Fourth Workshop on the Next Generation of CASE Tools*, pp. 35–51, Paris, France, June 1993.

[9] L.J. Campbell and T.A. Halpin, Abstraction techniques for conceptual schemas, in: R. Sacks-Davis, ed., *Proc. 5th Australasian Database Conference*, volume 16, pp. 374–388, Christchurch, New Zealand, January 1994. Global Publications Services.

[10] L.J. Campbell, T.A. Halpin and H.A. Proper, Conceptual schemas with abstractions – Making flat conceptual schemas more comprehensible, *Data & Knowledge Eng.* 20(1) (1996) 39–85.

[11] L. Cardelli and P. Wegner, On understanding types, data abstraction, and polymorphism, ACM Computing Surveys 17(4) (December 1985) 471–522.

[12] P.P. Chen, The Entity-Relationship model: Toward a unified view of data, *ACM Trans. Database Systems* 1(1) (March 1976) 9–36.

[13] P.N. Creasy and W. Hesse, Two-level NIAM: A way to get it object-oriented, in: T.W. Olle and A.A. Verrijn-Stuart, ed., *Proc. IFIP WG 8.1 CRIS-94 Conference on Methods and Associated Tools for the Information Life Cycle*, pp. 209–221, Maastricht, The Netherlands, September 1994.

[14] O.M.F. De Troyer, The OO-binary relationship model: A truly object oriented conceptual model, in: R. Andersen, J.A. Bubenko and A. Sølvberg, eds., *Proc. Third Int. Conf. CAiSE'91 on Advanced Information Systems Engineering*, volume 498 of *Lecture Notes in Computer Science*, Trondheim, Norway (Springer-Verlag, May 1991) 561–578.

[15] O.M.F. De Troyer and R. Janssen, On modularity for conceptual data models and the consequences for subtyping, inheritance and overriding, in: E.K. Elmagarmid and E.J. Neuhold, ed., *Proc. 9th IEEE Conference on Data Engineering (ICDE 93)*, pp. 678-685. IEEE Computer Society Press, 1993.

[16] O.M.F. De Troyer and R. Meersman, A logic framework for a semantics of object oriented data modelling, in: M. Papazoglou and Z. Tari, eds., *To appear in: Proc. the OOER '95 Conference* volume 932 of *Lecture Notes in Computer Science*, Gold Coast, Australia (Springer-Verlag, Dec. 1995).

[17] R. Elmasri and S.B. Navathe, Advanced data models and emerging trends, in: *Fundamentals of Database Systems*, 2nd edition, chapter 21 (Benjamin Cummings, Redwood City, California, 1994).

[18] R. Elmasri and S.B. Navathe, *Fundamentals of Database Systems*, 2nd edition. (Benjamin Cummings, Redwood City, California, 1994).

[19] R. Elmasri, J. Weeldreyer and A. Hevner, The category concept: An extension to the entity-relationship model, *Data & Knowledge Engrg.* 1 (1985) 75–116.

[20] D.W. Embley, R.B. Jackson and S.N. Woodfield, OO systems Analysis: Is it or isn't It? *IEEE Software* 12(3) (July 1995) 19–33.

[21] G. Engels, M. Gogolla, U. Hohenstein, K. Hülsmann, P. Löhr-Richter, G. Saake and H.D. Ehrich, Conceptual modelling of database applications using an extended ER model, *Data & Knowledge Engrg.* 9(4) (1992) 157–204.

[22] P. Feldman and D. Miller, Entity model clustering: Structuring a data model by abstractions, *The Computer J.* 29(4) (1986) 348–360.

[23] P.J.M. Frederiks, A.H.M. ter Hofstede and E. Lippe, A unifying framework for conceptual data modelling concepts, Technical Report CSI-R9410, Computing Science Institute, University of Nijmegen, Nijmegen, The Netherlands, September 1994.

[24] T.A. Halpin, *Conceptual Schema and Relational Database Design*, 2nd edition (Prentice-Hall, Sydney, Australia, 1995).

[25] T.A. Halpin and H.A. Proper, Subtyping and polymorphism in object-role modelling, *Data & Knowledge Engrg.* 15 (1995) 251–281.

[26] A.H.M. ter Hofstede, *Information modelling in data intensive domains*, Ph.D. Thesis, University of Nijmegen, Nijmegen, The Netherlands, 1993.

[27] A.H.M. ter Hofstede, H.A. Proper and Th.P. van der Weide, Formal definition of a conceptual language for the description and manipulation of information models. *Information Systems*, 18(7) (October 1993) 489–523.

[28] A.H.M. ter Hofstede and Th.P. van der Weide, Expressiveness in conceptual data modelling, *Data & Knowledge Engrg.* 10(1) (February 1993) 65–100.

[29] A.H.M. ter Hofstede and Th.P. van der Weide, Deriving identity from extensionality Technical Report CSI-R9416, Computing Science Institute, University of Nijmegen, Nijmegen, The Netherlands, December 1994. Electronically available as: ftp://ftp.cs.kun.nl/pub/SoftwEng.InfSyst/articles/IdentityExt.ps.Z.

[30] A.H.M. ter Hofstede and Th.P. van der Weide, Fact orientation in complex object role modelling techniques, in: T.A. Halpin and R. Meersman, eds., *Proc. First Int. Conf. on Object-Role Modelling (ORM-1)*, pages 45–59, Townsville, Australia, July 1994.

[31] G. Kappel and M. Schrefl, Local referential integrity, in: G. Pernul and A.M. Tjoa, eds., *11th Int. Conf. on the Entity-Relationship Approach*, volume 645 of *Lecture Notes in Computer Science*, Karlsruhe, Germany (Springer-Verlag, October 1992) 41–61.

[32] S.N. Khoshaflan and G.P. Copeland, Object identity, in: S.B. Zdonik and D. Maier, eds., *Readings in Object-Oriented Database Systems* (Morgan Kaufmann, San Mateo, California, 1990) 37–46.

[33] Y. Kornatzky and P. Shoval, Conceptual design of object-oriented database schemas using the binary-relationship model, *Data & Knowledge Engrg.* 14 (1994) 265–288.

[34] C.H.A. Koster, On infinite modes, *Algol Bulletin* 30.3.3 (1969) 86–89.

[35] X. Li and M.E. Orlowska, Conceptual modelling for complex objects, in: *Proc. Far East Workshop on Future Database Systems, pp.* 160–172, Melbourne, 1990.

[36] E. Lippe and A.H.M. ter Hofstede, A category theory approach to conceptual data modeling, Technical Report CSI-R9415, Computing Science Institute, University of Nijmegen, Nijmegen, The Netherlands, December 1994. Accepted for publication in *RAIRO Theoretical Informatics and Applications*.

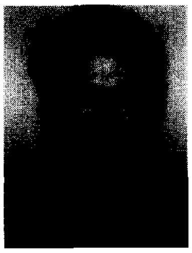
[37] G.M. Nijssen and T.A. Halpin, *Conceptual Schema and Relational Database Design*: A Fact Oriented Approach (Prentice-Hall, Sydney, Australia, 1989).

[38] A. Ohori, Orderings and types in databases, in: F. Bancilhon and P. Buneman, eds., *Advances in Database Programming Languages*, ACM Press, Frontier Series, (Addison-Wesley, Reading, Massachusetts, 1990) 97–116.

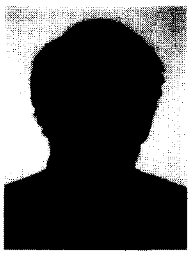
[39] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorenson, *Object-Oriented Modeling and Design* (Prentice-Hall, Englewood Cliffs, New Jersey, 1991).

[40] F. Saltor, M. Castellanos and M. Garcia-Solaco, Suitability of data models as canonical models for federated databases, *SIGMOD Record* 20(4) (1991) 45–48.

[41] P. de Sen and E. Gudes, A new model for database abstraction, *Information Systems* 7(1) (1982) 1–12.
[42] P. Shoval, Essential information structure diagrams and database schema design, *Information Systems*, 10(4) (1985) 417–423.
[43] M. Stonebraker, L.A. Rowe, B. Lindsay, J. Gray, M. Carey, M. Brodie, P. Bernstein and D. Beech, Third-generation database system manifesto, *SIGMOD Record*, 19(3) September 1990) 31–44.
[44] J.D. Ullman, *Principles of Database and Knowledgebase Systems*, volume I, chapter 3 (Computer Science Press, Rockville, Maryland, 1989).
[45] D. Vermeir, Semantic hierarchies and abstractions in conceptual schemata, *Information Systems* 8(2) (1983) 117–124.
[46] Th.P. van der Weide, A.H.M. ter Hofstede and P. van Bommel, Uniquest: Determining the semantics of complex uniqueness constraints, *The Computer J.* 35(2) (April 1992) 148–156.
[47] A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey L.T. Meertens and R.G. Fisker, *Revised Report on the Algorithmic Language ALGOL 68* (Springer-Verlag, Berlin, Germany, 1976).
[48] J.J.V.R. Wintraecken, *The NIAM Information Analysis Method: Theory and Practice* (Kluwer, Deventer, The Netherlands, 1990).



H.A. (Erik) Proper is currently a lecturer at the School of Information Systems from the Queensland University of Technology, Brisbane, Australia. He is a member of the Cooperative Information Systems Research Centre from that University. He is also a member of the Distributed Systems Technology Centre (DSTC); one of the Cooperative Research Centres funded by the Australian government, Australian Universities, and a number of multinational companies. Dr. Proper received his Master's degree from the University of Nijmegen, the Netherlands in May of 1990, and received his PhD from the same University in April 1994. In his Doctoral thesis he developed a theory for conceptual modelling of evolving application domains, yielding a formal specification of evolving information systems. From May 1994 to September 1995 he worked as a Research Fellow at the Computer Science Department of the University of Queensland, Brisbane, Australia. During that period he also conducted research in the Asymetrix Research Lab at that University for Asymetrix Corp, Bellevue, Washington. Dr Proper has co-authored several journal papers and conference publications. His main research interests include, federated information systems, evolving information systems, information retrieval, Object-Role Modelling, conceptual modelling in general, linguistic applications to conceptual modelling, World Wide Web and knowledge based systems.



P.N. Creasy received his B.Sc. in mathematics from the University of Adelaide and his Ph.D. in computer science from the University of Queensland, Australia. He has previously worked at the Australian National University where his research included investigating efficient implementation of deductive database systems. He is currently in the information systems group in the Computer Science department, University of Queensland where he is senior lecturer. His current research interests include data modelling, constraint representation and enforcement and knowledge representation. He is a member of IFIP WG8.1.