R.J. Wieringa

Axiomatic Specification of Database Domain Statics

Rapport nr. IR 133
oktober 1987

VRIJE UNIVERSITEIT AMSTERDAM

Faculteit Wiskunde en Informatica, De Boelelaan 1081, 1081 HV Amsterdam
Axiomatic Specification of Database Domain Statics

Roel Wieringa

Department of Mathematics and Computer Science
Vrije Universiteit
Room U4.37
Postbox 7161
1007 MC Amsterdam

uucp: roelw@cs.vu.nl
PRELIMINARIES

Chapter 1. Introduction
1.1 The universe of discourse and the domain
1.2 An extension of the four-level architecture

Chapter 2. Object identity
2.1 Identities as surrogates
2.2 Identities as sets
2.3 Domain axiomatizations
2.4 Identity axioms

Chapter 3. Aggregation
3.1 Attributes
3.2 The example domain
3.3 Attributes, properties and operations

Chapter 4. The generalization hierarchy
4.1 Types and kinds, natural and unnatural
4.2 Attribute inheritance
4.3 Classifications

Chapter 5. Objects and Possible worlds
5.1 Objects
5.2 State spaces and projection functions
5.3 Worlds
5.4 Extensions

Chapter 6. Static integrity constraints
6.1 Admissible worlds and the language $L_5(CON, \tau)$
6.2 Examples

Chapter 7 Comparison with other approaches
7.1 Some problems from philosophical logic
7.2 Universal relation models
7.3 Non first normal form relations
7.4 "Semantic" data models
Chapter 8. Conclusion

8.1 Summary 78
8.2 Future research 78

References 79

Appendix 1. List of notations 87
Appendix 2. Zermelo-Fraenkel set theory 88
Appendix 3. The example domain axiomatization in $L_3(CON)$ 95
Appendix 4. A model for the example domain axiomatization 97
Preface

In the past ten years, much work has been done to add more structure to database models than what is represented by a mere collection of flat relations (Albano & Cardelli [1985], Albano et al. [1986], Borgida et al. [1984], Brodie [1984], Brodie & Ridjanovic [1984], Brodie & Silva [1982], Codd [1979], Hammer & McLeod [1981], King [1984], King & McLeod [1984], [1985], Mylopoulos et al. [1980], Smith & Smith 1977a & b). The informal approach which most of these studies advocate has a number of disadvantages. First, a recent survey of some of the proposed models by Urban & Delcambre [1986] reveals a wide divergence in terminology and concepts, making comparison of the expressive power of these models difficult. Second, undefined or even ill-defined concepts are a hindrance, not an aid, for the analysis of the Universe of Discourse (UoD). Third, informal treatment of such complex structures as set hierarchies, generalization hierarchies and aggregation hierarchies all in one model, with some dynamics thrown in for good measure, bodes ill for the consistency of these theories.

The first goal of the research reported on is to integrate the static structures which these models propose in one coherent, axiomatic framework. It will be shown in chapter 7 that the theory presented here provides the needed conceptual foundations for these models. A second aim is to provide a possible worlds framework onto which to graft theories of the dynamics of the UoD. The third aim is to provide clear concepts which can aid the database model designer in his or her thinking about the UoD. In this report we concentrate on the first goal only, leaving the formulation of theories of domain dynamics and the application to system development as research goals for the near future.

The structure of this report is as follows.

Chapter 1 provides the necessary context for the theory by defining a four level structure for information systems (IS's). The goal of this and of forthcoming reports can then be stated in terms of this IS structure.

Chapter 2 formalizes the concept of object by combining ideas from database theory (surrogates and identities), philosophical logic (identity and rigid designation), axiomatic set theory (the hierarchy of sets) and systems theory (state spaces and state transition functions).

Chapter 3 defines attributes and introduces an example database domain. It also draws an important distinction between attributes and operations.
Chapter 4 investigates the lattice structure of the specialization/generalization hierarchy. The concepts of kind and type are defined, and criteria are given for a kind (type) to be natural.

The results of chapters 2-4 are distilled in a number of axioms about the UoD, some of which are common to all UoD's and others which are specific to individual UoD's. For the example UoD, these axioms are summarized in appendix 3. Appendix 4 gives a formal model for these axioms.

Chapter 5 defines the concepts of state and combines it with the concept of identity to define objects. A possible world is then defined as a function which assigns one state to each different object identity. Different possible definitions of object existence are discussed.

Chapter 6 treats the set of possible worlds as a set of possible models for static integrity constraints. Worlds which satisfy the static constraints are called admissible.

Chapter 7 uses the conceptual apparatus developed in chapter 2-6 to analyze the structures of "semantic" data models like TAXIS, RM/T, SDM and others. Two other models which stand in the relational tradition, the universal relation model and several proposals for non-first-normal-form models, are investigated as well. Also, a brief comparison with some problems and their solutions in philosophical logic, relevant to the specification of data models, is given.

Chapter 8, finally, summarizes the main results and lists topics for future research.

The appendices contain a list of notational conventions, an overview of ZF, a summary of the axioms for the example UoD used in this report and a description of a formal model for these axioms.

Thanks are due to Jan Willem Klop, Reind van de Riet and Hans Weigand for their careful reading of different versions of this report.
Chapter 1
Introduction

1.1. The universe of discourse and the domain

1.1.1.

In Griethuysen [1982], the Universe of Discourse (UoD) of an IS is defined in two different ways as

1. "those things and happenings to which the common understanding of the represented information refers" (p. 1-2) and

2. "The collection of all objects (entities) that ever have been, are, or ever will be in a selected portion of a real world or postulated world of interest that is being described." (p. 1-4)

These definitions are different because 1 can be taken to refer to past, present and future actual and possible entities, whereas 2 only refers to past, present and future actual entities. In this report we follow the first definition and define the UoD as follows.

The UoD is the collection of entities which are, have been, or will be actually or possibly of interest.

There are two important elements in this definition, viz. the references to 1. interestingness and to 2. actual as well as possible entities in the UoD. We discuss the first aspect in the rest of this section. The second aspect is subject of the rest of this report.

1.1.2.

The consequences of the statement that the UoD consists of entities of interest can be summarized under five headings.

1. The UoD is a social world. (At least the UoD's we are interested in.) It presupposes a being for whom there are entities of interest. In philosophical terms, the UoD presupposes an intentional being, i.e. a being who can direct his or her attention to an (abstract of concrete) entity and for whom the entity is of interest ("intentional" is here used with a meaning different from "intending" in everyday language). Many UoD entities only exist by virtue of the intentionality of human beings. For example, bank accounts, orders, financial obligations, letters of intents, etc. are entities which exist because people attach meaning to them. They are social entities (see Lehtinen & Lytinen [1986] for an analysis of IS’s as formal mechanisms for the representation of intentional entities, based upon the work of Searle [1969] and Searle & Vanderveken [1985]). If we remove the people from the UoD, intentional entities like bank accounts, obligations etc. disappear as well.

2. The perception of the UoD often is part of the UoD. This property of the social world is argued in Searle [1984] and is actually implicit in the above point. The common understanding of a bank account is part of the universe of bank accounts. It must be stressed
that a social world is a world of intentional beings and vice versa.

3. *The perceptions of the UoD differ.* The initial stages of information analysis typically consist of talking to people living in the UoD and reading documents describing the UoD. Virtually no source of information about the UoD gives the same description of what the UoD is. One may wonder if, indeed, there is "a" UoD. (Kent [1980] gives numerous illustrations of differences in views of the UoD and the problems encountered in formalizing the UoD.) On the one hand, people living in the UoD have different ideas about what the reality is they live in but on the other hand, they manage to talk to each other, which presupposes a common reality.³

4. *Conflicts exist at any (un)desirable metalevel.* The analyst typically uncovers political, psychological, sociological and economical tensions which must be considered to be part of the UoD as well. Different views of the UoD lead to conflicts, but reflecting on these conflicts, different participants in the conflicts have different views of what these conflicts are about, where they come from, how they should be resolved, etc.

5. *The introduction of the IS changes the UoD.* I behave differently if I know my working time is registered. Instead of going to the post-office on the way to work, I will go to work first, register, and then go to the post-office. Registration is the basic function of any IS. Registration can be perfected to such a large extent that it becomes permanent observation (by video monitors, electronic identification cards etc.), and it is a known fact from social science that permanent observation fundamentally changes the nature of human behavior. In particular, it tends to diminish the play-like, creative aspects of human behavior.

We can divide system development into roughly two parts, the analysis of the UoD and the design of an automated IS. Human intentionality is the source of all complexity in the first half of system development, the analysis of the UoD (machine unwieldiness being the source of all headaches in the second half). The means to attack this complexity lie in the very source of the problem: Intentionality itself. The system analyst is an intentional being as well who, in principle, is capable of understanding what is going on and who has (or ought to have) the linguistic capability to verbalize it. This explication process moves in a circle running from a grasp of the whole of the UoD to a grasp of the details and back again. In philosophy and linguistics this is known as the *hermeneutical circle*, after the problem of understanding a difficult text from a foreign culture far removed from the reader in time and space. Understanding of the parts of the text presuppose understanding of the whole and vice versa (see also Winograd & Flores [1986]). It is in this intentional context that we explicate the functions of the IS. Again, understanding of the part of the UoD to be modelled by the IS presupposes an idea of the functions which the IS is to fulfil, but those functions can only be understood in terms of the part of the UoD to be modelled.

---

³ This suggests that philosophical idealism (the world exists in our minds only) and realism (our abstract concepts refer to real entities) are only two halves of a truth. Philosophical idealism is a station long past in the history of philosophy, but seems to be an easy refuge for mathematically oriented systems analysts for whom everything is relative and the UoD is what the customer thinks it is.
1.1.3.

I make the drastic move to abstract from all problems related to intentionality, both in the UoD itself as well as in the first half of system development. This is done by defining the domain of an IS as an abstraction of the UoD in which at least all intentionality has been removed.

The domain is the collection of all objects which are, have been, or will be actual or possible.

The domain can be viewed as the explicit description of the relevant aspects of the UoD which is the output of the first stage of information analysis and the input to the second stage, designing an (usually automated) IS in a finite medium. At the domain level of abstraction, all conflicts have been resolved (or, more likely, eliminated, ignored, or suppressed). The domain is Nagel's [1986] "view from nowhere". By placing it at the interface between the informal and formal stages of system development, we recognize that both stages deal with a real part of the world.4

The domain may also be construed as the UoD as viewed by a disinterested all-knowing Observer with infinite computational power. This is the universe under the eye of Eternity, the machine which 17th-century philosophers thought the world to be. In this Platonic heaven, all objects exist for all eternity (this will be formalized below). Existence of the objects in the domain independent of any human observer is the hallmark of Platonism (Bernays [1935]).

1.1.4.

To aid intuition in reading the following chapters, I already sketch the domain structure in an informal manner. The domain is a directed graph in which the nodes are possible worlds and the arcs are transitions between possible worlds. Each possible world may be thought of as a domain state. Alternatively, the set PW of possible worlds may be viewed as a state space. The arcs in the directed graph of possible worlds then represent a state transition function. There is one trajectory through the state space which represents the actual life of the UoD. I do not distinguish a current domain state. The Observer has infinite knowledge of the domain and possesses a description of the complete graph.

Each possible world is a set of objects which are called actual in that world. Each object is a pair (identity, state). The identity of an object is immutable and persists through change. It is also unique across possible worlds and thus serves to identify an object uniquely, independently of its state. The state of an object is a tuple of object identities. For example, to represent a UoD entity named John with address 17, Fleet Street, we may have

\[(p_1, (\text{John}, a_1))\].

This is a domain object with identity \(p_1\) and state vector \((\text{John}, a_1)\). \text{John} is the identity of only one object, \((\text{John}, \lambda)\) and \(a_1\) identifies the object

4. Programmers (and not programmers alone) tend to think that what is not formalizable, explicitly describable, does not exist. Nagel calls this the "epistemological criterion for reality" and points out that it is a remnant of (philosophical) idealism.
(a), (17, Fleet Street)).

The two components of this state vector identify the objects (17, ()) and (Fleet Street, ()). Objects are thus aggregated out of other objects, and, going downward, the aggregation stops at identities like 17 and John, which may be called atoms. Atoms have a zero-length state vector. They have no internal state and may be considered to be equal to the (only) object they identify.

All of this will be treated in greater detail below.

1.2. An extension of the four-level architecture

1.2.1. Several researchers, among whom Kent [1980], have proposed a four-level architecture for IS's. Between the function level (the "external schema") and the implementation level (the "internal schema"), Kent proposes to insert two levels, called the enterprise and the collective level. The enterprise level describes the UoD, whereas the collective level is an implementation-independent description of the DB shared by all application programs (hence "collective"). We adopt this approach but change the usual representation of the architecture for two reasons. First, we want to account for the fact that a computer is essentially a simulation device. The reason why a universal Turing machine is a good abstraction from a computer is that the UTM possesses the essential property of a computer in bare form, viz. that it is universal, or, in other words, that it can simulate any machine. I described the domain as an infinite world machine. It is the job of a finite computer to simulate the domain as well as it can.

The second reason for changing the representation of the four-level architecture is that we want to show that the user, as part of the UoD, looks at the UoD when he or she interrogates the DB. Abstracting from differences in opinion between users, the user looks at the domain but gets answers from the DB. We thus get the picture of figure 1. The UoD is actually a number of UoD views of people who participate in the UoD. The dots around the UoD suggest that the UoD is, in general, the place of ambiguous and informal processes.

After system analysis, the UoD is abstracted into an unambiguous domain in terms of which a number of user domain views are defined. The domain and its views are explicitly described, whereas the UoD isn't. The solid lines around the domain and other parts of figure 1 represent the fact that these are explicitly described. The arrows from the UoD to the domain suggest that the UoD can be explicated in different ways to a domain.

The database is a finite representation of the domain and the database views finitely represent the domain views. The transition from the domain to the database is a reduction, for the domain is, in general, infinite, whereas the DB is always finite. The arrows from the
Figure 1.
domain to the DB represent the fact that one DB state does not unambiguously represent one state of one domain. Each DB state is compatible with states of different domains (so that there is, in general, room for difference in opinion about the meaning of the result of a DB query). Even if the DB were unambiguously compatible with only one domain, each DB state will in general represent many domain states.

The DB and its views are simulated by an implementation in which actual hard- and software have been chosen to do the job.

1.2.2.

Kent's four-level architecture appears in figure 1 as the implementation ("internal schema"), database, ("collective schema"), domain, ("enterprise schema"), and database views, ("external schemas"). The extension of the four-level approach thus consists of the addition of domain views and a more precise definition of the relation between UoD and domain and between domain and DB.

The domain corresponds roughly with what is called a "conceptual model" or "semantic model" in systems like TAXIS, SDM and others. My criticism of those models can now be rephrased as follows.

1. They do not provide concepts which are clear enough to help in the abstraction from the UoD.
2. They use different, incompatible, undefined or even ill-defined concepts to describe domain structures.
3. They do not provide formal tools to guarantee that the domain specification is consistent.
4. They do not provide the conceptual tools to explicate the decisions which must be made to reduce an infinite domain to a finite database.

To this a fourth critique can be added,

The subject of this report is the static structures found in the domain. Its goal is to axiomatize static domain structures and provide foundations for other approaches to static domain structures. Forthcoming reports will add dynamic structures, apply these to a non-trivial task of UoD abstraction, and study the issues of domain reduction.
Chapter 2
Object identity

2.1. Identities as surrogates

2.1.1.

The identity of a domain object \((s, o)\) is a surrogate for the identity of a UoD entity. Hall et al. [1976] cite Langefors [1966] as saying that a UoD entity "may be regarded as being a simple bundle of property values associated with it." This view has several problems, notably that the set of values stored for an object may not identify it uniquely. This forces us to introduce an artificial property, the key of the object, which is unique for all domain objects. A key is thus what in logic is called a uniquely identifying description of domain objects. However, with keys as with uniquely identifying descriptions, a number of problems remain. A key may cease to be uniquely identifying, or there may be different keys used by different people, or a key may be changed. For example, a unique personel number may be replaced by another unique personel number, and this replacement is indistinguishable from the dismissal of an employee and the hiring of another one with the same non-key properties and a new personel number. Hall et al. therefore introduce the concept of surrogate as the unique representative in the IS of a UoD entity. The surrogate of an UoD entity is a system-generated, unchangeable bearer of properties. By contrast, any combination of one or more properties borne by the surrogate may be used as a key.

In terms of domains, the identity of a domain object is an abstraction of the identity of a UoD entity and bears properties which are in their turn abstractions of identities of UoD entities. Conversely, the identity of a domain object represents the identity of a UoD entity, and its properties represent properties of the UoD entity. Specifically, a surrogate represents the following information about a UoD entity:

1. that the entity exists actually or possibly in the past, present or future of the UoD,
2. that it is different from other entities in the UoD and
3. that it persists through time.

7. On the surface, this is close to the Buddhist standpoint that things are just bundles of momentaneous properties which join together and fall apart again in each instant and have no identity over and above their temporary association. It is also close to Hume’s standpoint that we form ideas of things by habitually forming associations of elementary impressions. Closer investigations, though, reveals more differences than parallels. See E. Conze, "Spurious Parallels to Buddhist Philosophy" in Conze, Buddhist Studies 1934-1972, Wheelwright Press, n.d., 229-242 and "Buddhist Philosophy and its European Parallels," ibid., 210-228.

8. These are the three characteristics of what the Buddhists call avidya, common sense knowledge which enables us to get by in the everyday world. \textit{Avidya} is usually translated as "ignorance." It is contrasted with \textit{prajna} (translated as "intuition"), which is pre-reflexive awareness of the unsubstantial and fleeting nature of this world. \textit{Prajna} makes us realize the relativity of the three statements concerning surrogates (we don’t even negate them). To be precise, using our \textit{prajna} we see that 1. it is wrong to say that a thing exists through time separately from other things, 2. that it is wrong to deny this fact, 3. that it is wrong to both affirm and deny it and 4. that it is wrong to neither affirm or deny it. In other words, using our \textit{prajna}, we see that entities have no identity. The absolute truth seen in \textit{prajna} is unutterable, unteachable, etc. (T.V. Murti, The essential philosophy of Buddhism, p. 239 ff.). Clearly, it is the avidya of the domain we want to represent in a knowledge base.
2.1.2.
There is some conceptual clarification to be done concerning the status of abstract entities like numbers, strings, or Boolean values. Having so carefully distinguished the domain from the UoD, I must now note that numbers -assuming that they exist- exist in the UoD. But as abstract entities they exist in the domain as well. One wonders if numbers would exist if people wouldn’t, or if the existence of numbers presupposes the existence of an infinite Observer in Whose mind numbers (and we) exist as a cosmic dream (Berkeley). These problems won’t bother us at all in this report. But this brief remark does show, in my opinion, that database modelling is close to the “world knot” (Schopenhauer).

2.1.3.
The advantages of the use of surrogates in IS’s have been discussed, among others, by Atkinson et al. [1983], Codd [1979], Date [1983], Hall et al. [1976], Kent [1979], and Khoshafian & Copeland [1986]. We summarize them below, where we distinguish the advantages for use in the domain and its views, in the database and its views, and in the implementation. Most advantages follow from the independence of identity from the state of the object.

1. In the **domain**, we have **identity through domain change**. This makes it possible to give meaning to the phrase “same object” even if all attribute values have changed. In particular, keys may be changed, e.g. when the domain is reorganized.

2. In **domain views**, we have **identity under different views of the domain**. It makes it possible to speak of the same entity even when two totally different descriptions of the entity are offered. For example, different departments in an organization may use different keys, or before and after an organization different keying schemes are used.

It is by these two advantages that we are able to use surrogate names as rigid designators for objects, i.e. as names which refer to the same object in every possible domain state (cf. Kripke [1971], [1980]).

3. The use of surrogates in the **database** has advantages which follow from the fact that identity is independent of the spatial or temporal organization of the DB. The points below parallel the advantages for use in the domain in that they concern identity (spatial uniqueness) and change (temporal persistence).

3.1. **Persistence.** The information that a domain object exists survives database transactions. This facilitates a uniform treatment of stored and derived information. Derived information concerns persistent entities, but happens to be computed by the implementation every time it is needed.

3.2. **No duplications.** The information that domain object exists is represented once. Referential integrity constraints are built into the notation introduced below.

3.3. **Indistinguishable objects.** Two objects may be allowed to be different even when they agree on all attribute values represented in the database. This is not allowed in relational databases, where all tuples in a table must differ in at least one attribute value.

4. For **database views**, the advantages follow from the fact that all information about an object is attached to a single identity.
4.1. Thus, the entity join (Kent [1979]) is built into the domain. All information about a domain object is represented in a single state vector which is attached to the identity of the object. There is no need for the user to join tuples from different relation instances together in order to get information about a single object.

4.2. Moreover, the use of surrogates allows queries to be expressed as path-expressions which are simpler than the corresponding queries in a relational language using keys to identify information about a single object in different relation instances (Khoshafian & Copeland [1986]).

5. In the implementation, surrogates are roughly global record identifiers in virtual address space. This allows the separation of identity from addressability. Independence of object identity from location in virtual and hence physical address space facilitates sharing of objects among different programs. In appendix 4 a model of the formal theory of the domain is given which is isomorphic to \( \mathbb{N} \). The set of natural numbers in this model serve as global identifiers in an infinite DB.

2.1.4.

It is worth dwelling at some length on these advantages, for surrogates carry disadvantages as well. Two phenomena of domain dynamics seem to call for an adaption of the idea that a surrogate is the identity of a single object in all possible domain states.

1. **Merging.** Two entities may merge into one. For example, part-time jobs, budgets, committees, projects etc. may be merged. What happens with their identity during the merge?

2. **Splitting.** The converse problem, of course, is the splitting of one entity into several.

Codd [1979] proposes a coalesce operator which merges surrogates. Before such operators are contemplated, less exotic forms of change should be tackled. For example, in the presence of these two phenomena, the principle of the substitutivity of identicals is violated (Hintikka [1969]).

A number of other problems with surrogates have been mentioned in the literature (e.g. Reiter [1984] and Levesque [1984]). These all have to do with the occurrence of noise in the communication channel from the domain to the DB or with the finiteness of the DB. We can distinguish the following problems.

1. **Disjunctive information.** An attribute value may be one out of a group of surrogates, but we do not know which because this information has been lost in the communication from the domain to the DB. For example, a color may be either red or brown (but not both).

2. **Conjunctive information.** The converse problem is that the message arrives at the DB undistorted, but the DB cannot be in sufficiently many different states to represent the information. For example, a color may be red-brown, but the DB only disposes of the attribute values red and brown. Both values are equally right (or wrong). Consequently, the value red in the DB represents red and red-brown in the domain, and similarly for brown. The problem can be solved by using two values, (red, brown), which in conjunction represent a single domain value. This must be distinguished, of course, from the case where only one of the two DB values is correct, and from the case where there are genuinely two different domain objects to be represented (e.g. a project consisting of two
members).

3. **Errors.** Even if the discriminating ability is sufficient, there may be errors. For example, two DB surrogates may turn out to represent the same DB object, or one may represent two objects, or one DB surrogate may represent the wrong domain object.

2.1.5.

Having relativized the surrogate concept somewhat, we note that none of the problems mentioned so far have been solved satisfactorily, let alone that a single solution to all of them is known. Because the surrogate idea is particularly suited to dealing with identity through change, I will follow up the idea to see how far I can get with it.

The upshot of this section is that we have a set $S$ of surrogates for the identity of all possible UoD entities. We declare the concept of identity of a UoD entity understood (see Kripke [1971], [1980]) and assume that it is applicable to the relevant entities in the UoD. To distinguish the theory developed in this report from other theories (relational, object-oriented) I call it the ABSURD model, for ABstract SURrogate Domain.

2.2. Identities as sets

2.2.1.

Since Makinouchi [1977] the need for non-atomic attribute values has been recognized in the relational DB community. For example, project members, several copies of the same book in a library, the children in a family are real-world entities which are naturally represented in the domain by sets. A number of researchers have studied the integration of set-valued attributes in the relational model, in an object-oriented model or in a logic programming framework (see section 7.3 for references and a discussion). A domain object has the form

$$(s_0, (s_1, \ldots, s_n))$$

where $s_i \in S$, $i = 0, \ldots, n$. If we allow sets as attribute values, then any $s \in S$ may be a set. So we ought to allow sets as object identities as well. For example, in the ubiquitous project-employee UoD, we may be interested in employees as well as sets of employees. Then in the abstract domain representing this UoD, we may have a surrogate $s_0$ which represents (the identity of) a set of employees whose identities are represented by $s_1, \ldots, s_n$. The $s_0$ is the same set as $\{s_2, \ldots, s_n\}$.

$s_0 = \{s_2, \ldots, s_n\}$.

The $=$ is equality of sets, defined in axiomatic set theory.

Now, which surrogates are sets of which other surrogates? The Observer knows, but we don't. The easiest way to think of $S$ is to see it as a subset of the class $V$ of all possible sets. The precise structure of $V$ depends upon the particular axiomatization of set theory one works in. Nothing is assumed about $V$, except that all its elements are sets. Now, if $S \subseteq V$, then any $s \in S$ is a set. There are three cases:

1. $s \subseteq S$. We are interested in the elements of $s$, so they are identities of domain objects. An example is the set of employees above. Surrogates of this type may be called,
misleadingly, _set_ surrogates.

2. \( s \cap S = \emptyset \). We are interested in no element of \( s \). An example is a surrogate in an agricultural domain representing a crop growing on a field in a particular period of time. The crop is a set of individual plants, but in the domain we represent only the identity of the whole crop, not of its elements. Surrogates of this type are at the bottom of the set hierarchy in \( S \) and play the role of _Urrelemente_, _primitive elements_, in the domain.

3. If neither 1 nor 2 is the case, we may speak of set surrogates as well. An example is the representation of a printing of a book in a library domain. The library may possess several copies of the book of the same printing. The printing is represented in the library domain as a set, and a few of its elements are represented in the domain as well.

Note that we now distinguish two hierarchies, each with its basic elements. The smallest elements in the aggregation hierarchy are called atoms, the smallest elements in the set hierarchy are called primitive elements.

2.2.2.

Before these ideas are formalized, I want to point out the usefulness of this view of surrogates and object identity. First, we can in the formal language defined below meaningfully apply the binary predicate \( \in \) (in infix notation) to surrogates:

\[ s_1 \in s_2 \]

is either true or false in an interpretation of this language. This simplifies the description of the domain greatly.

Second, wildly varying _UoD_ structures can be represented by domain objects in a concise, uniform manner. The orthogonal classifications of primitive versus non-primitive surrogates and atomic versus non-atomic surrogates gives rise to the following four possibilities.

1. \((s_0,())\) with \( s_0 \) a primitive, atomic surrogate. For example, \((John,())\).

2. \((s_0,())\) with \( s_0 \) a non-primitive, atomic surrogate. For example, the set of employees on a project is \((\{s_1, ..., s_n\},())\) (assuming that we don't find any attributes of this set relevant enough to represent in the domain).

3. \((s_0,(s_1, ..., s_n))\) with \( s_0 \) a primitive, non-atomic surrogate. This is the traditional structure for domain objects. For example, \((s_0,(12,John))\).

4. \((s_0,(s_1, ..., s_n))\) with \( s_0 \) a non-primitive, non-atomic surrogate. For example, \((\{s_1, ..., s_n\},(s_{n+1}, s_{n+2}))\) where \( \{s_1, ..., s_n\} \) is a set of identities of employees and \((s_{n+1}, s_{n+2})\) is a pair \((\text{average- age}, \text{average- salary})\). (There are obvious integrity constraints on these attribute values and attribute values of the employees in the set. Integrity constraints are treated in chapter 6.)

Compare this with the objects of Khoshafian & Copeland [1986], which have the form \((\text{identifier, type, value})\), where _type_ can be one of _atom_, _set_, and _tuple_. Because in my model _identifier_ can be a set and a tuple is a tuple of identifiers, I can drop the _type_ and let _value_ be of _tuple_-type only. The bonus is a simpler object structure and a more powerful expressiveness. Objects are basically systems with a state space and an identity so that we can keep isomorphic systems apart. (In a forthcoming report I introduce a state transition function for objects.) Case
4 above, which has been called a "metaclass" in some modelling approaches, cannot be represented by Khoshafian & Copeland.

As will be shown in section 7.4, the use of axiomatic set theory allows one to clear up a conceptual muddle surrounding "metaclasses", "associations" and "higher-order classes" in some "semantic" data models.

2.3. Domain axiomatizations

2.3.1. I now formalize the foregoing ideas by introducing a formal language \( L_S \) to describe the static structure of the domain. Figure 2 shows the different levels of abstraction involved in the formalization.

![Figure 2](image)

The domain is formalized as a directed graph \( PWG \) of possible worlds. \( PWG \) is an abstraction of the UoD and represents the UoD in an abstract world. \( PWG \) is a subset of the class \( V \) of all sets, as is \( S \). The elements of \( S \) are named by \( s_i \). \( S \) and \( s_i \) are introduced below as constants of a set-theoretical language. Other constants will be introduced as well. \( PWG \) is a defined term, formally part of the metalanguage in which we talk about \( L_S \) but practically a constant in \( L_S \).
2.3.2. Definition

A static domain language $L_5(\text{CON})$ is a first-order language using the following symbols.

1. **Individual variables:**
   There are infinitely many variables. I do not actually give the names of the individual variables.

2. **A set $\text{CON}$ of constants:**
   
   $$\text{CON} = \text{CON}_A \cup \text{CON}_NK \cup \{\text{NA}, A, NK, em, s\}$$
   with
   
   $$\text{ext} \in \text{CON}_A,$$
   $$\text{Ext} \in \text{CON}_NK,$$
   $$S \in \text{CON}_NK.$$

   The meaning of the special constants $\text{NA}$ etc. is explained when needed. The constants in $\text{CON}_A$ are called *attribute names* and those in $\text{CON}_NK$ are called *natural kind names*.

3. **Predicate symbol:** $\in$.

4. **Logical symbols:** $\neg, \land, \lor, \Rightarrow, \Leftrightarrow, \forall, \exists$.

5. **Auxiliary symbols:** $(,), [ ]$.

   The formulas of $L_5$ are built according to the usual rules. I use, among others, the letters

   $$x, y, z, ...,$$

   as metavariables over the collection of individual variables. (See appendix 1 for a complete list of different types of metavariables.)

   The following metavariables over $\text{CON}$ are used.

   $$c_0, c_1, ..., \text{ are metavariables over } \text{CON},$$
   $$a_0, a_1, ..., \text{ are metavariables over } \text{CON}_A, \text{ and}$$
   $$\underline{k}_0, \underline{k}_1, ..., \text{ are metavariables over } \text{CON}_NK.$$  

   Occasionally, I use $t$ as metavariable over terms (i.e. individual variables or constants). The reason for underlining the metavariables over $\text{CON}_A$ and $\text{CON}_NK$ is that I need the letters $a$ and $k$ for metavariables over the individual variables which range over the corresponding sets. An example of sets of constant symbols is

   $$\text{CON}_A = \{\text{ext}, \text{name}, \text{emp#}, ...\},$$
   $$\text{CON}_NK = \{\text{Ext}, \text{Emp}, \text{Secr}, ...\}.$$  

   Then $\underline{a}$ denotes any element of $\text{CON}_A$ and $\underline{k}$ denotes any element of $\text{CON}_NK$. Note that attribute names are written in small letters and natural kind names are written with an initial capital letter. The intuitive meaning of the constants in the example strongly suggest that we are talking about a domain consisting of employee names and employee numbers. These intuitions are not wholly accidental, but it must be stressed that they are only warranted in so far as they are formalized by the axioms to follow.

   The letters
\[ \phi, \psi, \eta \]

are metavariables over the set of well-formed formula's (wff's). For example, the formula
\[ \forall x [\phi(x)] \]
is an expression in the metalanguage (English), from which a formula in \( L_{ZF} \) may be gotten by replacing \( \phi \) by a string of the symbols listed above and \( x \) by an actual variable of \( L_{ZF} \). I follow the convention that in \( \phi(x) \) at least \( x \) occurs as free variable.

In the sequel, for ease of writing I use \( L_S \) instead of \( L_S(CON) \). I also assume that we are talking about a particular domain language of a particular abstraction of a particular UoD, so that it makes sense to talk of the" domain language \( L_D \).

**2.3.3. Definition**

The symbol
\[ \{x \mid \phi(x)\} \]
is called a class symbol.

A generalized term is a term or a class symbol. A generalized atomic formula has the form
\[ t_1 \in t_2 \]
where \( t_i \) are generalized terms. A generalized formula is built up according to the usual rules from generalized atomic formulas.

For example,
\[ c \in x, \]
\[ x \in y, \]
\[ x \in \{y \mid \phi(y)\}, \]
\[ \{x \mid \phi(x)\} \in c, \text{ and} \]
\[ \{x \mid \phi(x)\} \in \{y \mid \psi(y)\} \]
are generalized formulas. All generalized formulas can be translated into formulas (Takeuti & Zaring [1971]). For example the last generalized formula is an abbreviation of
\[ \exists y [\psi(y) \land \forall z [z \in y \Rightarrow \phi(z)]] \]

For brevity, I drop the qualification "generalized" from now on and use the qualification "restricted" instead to indicate that I am not talking about generalized terms etc.

**2.3.4.**

\( L_S \) is a set-theoretical language, which means all functions and predicates must be defined in terms of \( \in \). I simply assume any relevant mathematical predicate or function to be defined in \( L_S \), free for use in our formal statements about the domain. Thus, I will freely talk about sets like \( x \times y \) (the cartesian product of \( x \) and \( y \)) and \([x \rightarrow y]\) (the set of functions from \( x \) into \( y \)) without bothering about their definition. A function \( f : x \rightarrow y \) is a set as well. Formally, \( f : x \rightarrow y \) is a three-place predicate on the sets \( f, x \) and \( y \). See Takeuti & Zaring [1971] for the definitions in ZF (Zermelo-Fraenkel set theory) and appendix 2 for a summary.
In virtually all domains, \( \mathbb{N} \subseteq S \). Accordingly, I assume the operations \(+\) etc. to be defined on \( \mathbb{N} \). If need be, we may as well assume the Peano axioms and real arithmetic to be applicable to our domain. This way, we are freed from reinventing wheels designed to cope with a restricted formalism and can concentrate on the precise definition of domain structures. At the domain level of abstraction we thus have the same flexibility we desire from DBMS's, viz. the ad hoc definition of functions and of sets of objects on which these functions are defined.

2.3.5. Definition
A standard structure for \( L_S \) is a pair \([V, I]\) where \( V \) is a class of sets and \( I \) an interpretation function such that
1. \( I(c) \in D \) for each \( c \) in \( CON \) and
2. \( I(\in) = \in \).

Thus, the constants are interpreted as sets in \( V \) and \( \in \) is interpreted as the relation "is element of" on \( V \). For \( V \), I take the universe defined by the axioms of ZF. This is not essential, the only condition on \( V \) being that all elements of \( V \) are sets, so that \( x \in y \) always evaluates to true or false, for \( x, y \in V \).

By interpreting \( \in \) as the "element of" relation, we assure that the definitions of functions and predicates have the usual meaning. 9

2.3.6. Definition
A value assignment in \([V, I]\) is a mapping \( \phi \) from the individual variables into \( V \).

2.3.7. Definition
The interpretation of a (restricted) term \( t \) in \([V, I]\), given a value assignment \( \phi \), is
\[
[t]_{\phi, I} = \phi(x) \text{ if } t = x,
[t]_{\phi, I} = I(c) \text{ if } t = c.
\]

\( = \) denotes syntactic equality.

2.3.8. Definition
A restricted atomic formula \( t_1 \in t_2 \) is true under value assignment \( \phi \) iff
\[
[t_1]_{\phi, I} \in [t_2]_{\phi, I}.
\]
We write \([V, I] \models_\phi t_1 \subseteq t_2 \).

---

9. I have now defined a framework in which the domain is an abstraction of the UoD in a set-theoretical abstract universe. This is reminiscent of the classical idea that the essence of the world lies in the abstract realm of numbers, the only difference being that we now know that sets are even more basic than numbers and that number theory can be reduced to set theory. The description of nature in the language of numbers can be traced back to the Pythagoreans, from where it reached modern science via the Neoplatonists and Copernicus. See T.S. Kuhn, The Copernican Revolution, Harvard UP [1975], pp. 128, 141 and A. Koestler, The Sleepwalkers, Pelican [1968], p. 201.
2.3.9. Definition

The interpretation of non-atomic formulas is defined in the usual way. \(\square\)

Note that generalized formulas receive an interpretation because they abbreviated restricted formulas.

2.3.10. Definition

\([V, I]\) is a **model** for a set of sentences \(\Sigma\) iff the sentences are true in \([V, I]\) for all value assignments. The set of all sentences true in \([V, I]\) is called the **theory** of \([V, I]\), written \(Th([D, I])\).

Let \(\Sigma_{FOL}\) be an axiomatization of first order logic. The set of all sentences which can be derived from \(\Sigma \cup \Sigma_{FOL}\) is called the set of **consequences** of \(\Sigma\), written \(Cn(\Sigma)\). An axiomatization of \([V, I]\) is called **sound** if \(Cn(\Sigma) \subseteq Th([V, I])\) and **complete** if \(Th([V, I]) \supseteq Cn(\Sigma)\). \(\square\)

In the following chapters, I give a set of axioms

\[\Sigma_S = \Sigma_{FOL} \cup \Sigma_{I} \cup \Sigma_{A} \cup \Sigma_{NK} \cup \Sigma_{SP}\]

which axiomatize the static part of the domain.

\(\Sigma_{FOL}\) is any axiomatization of FOL. One such axiomatization is given in appendix 2.

\(\Sigma_{I}\) is the set of **identity axioms**. They are given below.

\(\Sigma_{A}\) is the set of **attribute axioms or aggregation axioms** and is given in chapter 3.

\(\Sigma_{NK}\) is the set of **natural kind axioms** and is given in chapter 3 as well

\(\Sigma_{SP}\) is the set of **specialization axioms** and is given in chapter 4.

Jointly, they characterize the static structure of the domain as a set of possible worlds. Chapter 6 adds more knowledge in the form of **static integrity constraints**, which may be viewed as FOL formulas to be satisfied by a possible world to count as an **admissible world**. Because FOL inference rules are sound, we have

\[Cn(\Sigma_D) \subseteq Th([V, I])\]

All statements derivable from \(\Sigma_S\) are therefore true. Needless to say, the converse is not true.

2.4. Identity axioms

2.4.1.

The sentences in \(\Sigma_I\) are called the **identity axioms** of the domain theory. There are three identity axioms.

1. \(s : \mathbb{N} \rightarrow \text{onto} \ S\).
2. \(\emptyset \in S\).
3. \(em \in S\).

The set \(\emptyset\) is defined as the class \(\{x \mid x \neq x\}\). It represents the **absence** of a UoD entity. For example, if project \(s_0\) has no members then the appropriate component of its state vector will be \(\emptyset\). Similarly, an employee \(s_1\) without an address has will have \(\emptyset\) at the component of his
state vector representing his address. Incidentally, in this set-theoretical context $\emptyset = 0$. This emphasizes the role of the number 0, which is the indication of absence\textsuperscript{10}.

The constant $em$ stand for "existence monitor" and will be used in chapter 5 to distinguish actual from possible objects in a domain state.

Axiom 1 requires $S$ to be a countably infinite set of which the elements are named by $s$. I denote $s(i), i \in \mathbb{N}$, by $s_i$ and treat $s_i$ as a name for an identity in $S$. The axiom contains two requirements on the naming function $s$.

1. $s_i \neq s_j$ for $i \neq j$. This is equivalent to an infinite set of unique name axioms (Reiter [1984]). They demand that different identity names refer to different identities.

2. $\forall x \in S \exists i[x = s_i]$. This is Reiter's domain closure axiom. It says that all possibly relevant identities are named by $s$.

2.4.2. Definition

Let $x \in S$. If $x \cap S = \emptyset$, $x$ is called a primitive identity, otherwise it is called a set identity.

\textsuperscript{10} The symbol 0 comes from Hindu mathematics, which derived it from Buddhism, where it stands for emptiness, or sunya. Its use in European arithmetic can be traced back no farther than 1478, before which date emptiness was inconceivable (and arithmetic very difficult). See Hollingdale & Tootill [1975], p. 22 ff...
3.1. Attributes

3.1.1. Attributes

All information beyond the mere fact that certain entities exist is represented by attributes. I take a functional view of attributes, i.e. an attribute like age is a function from the set of persons to natural numbers. To facilitate the representation of a change in attribute value, I do not represent an attribute by a single function, but by a set of functions. This way, a change in attribute value is modeled by a replacement of one function in the attribute by another function in the same attribute. We thus maintain the information that the new value is a value of the same attribute.

Each function in an attribute is made into a total function by requiring that NA ∈ S and stipulating that for f ∈ a and x ∈ S, f(x) = NA if a does not apply to x. (NA stands for Not Applicable.) We thus take the idea of applicability as unexplicated primitive. The use of NA is not very essential but makes life a bit easier.

The properties of attribute inheritance follow in a simple way from this view of attributes. In the following definitions I use

k₀, k₁, ...

as metavariable over subsets of S. Thus, kᵢ names an individual variable which is a subset of S.

3.1.2. Definition

Let k₁ ⊆ S. A non-empty set

{[k₁ → k₂] = \{f \mid \exists f: k₁ → k₂∪\{NA\}\}}

is called an attribute. The set k₁ is called the domain of the attribute and k₂ is called the range of the attribute. An attribute is said to be applicable to the elements of its domain. □

Alternatively, an attribute [k₁ → k₂] may be viewed as a non-empty set of functions from k₁ into k₂. I use the symbols

a₀, a₁, ...

as metavariables over attributes. Remember that aᵢ is a metavariable over the set CONₐ of attribute names. The name of an attribute is intended to convey the role of the attribute. We have thus made one version of the universal relation assumption that each attribute has the same role independent of the context in which it occurs (see section 7.2 for a discussion).

By demanding that each function in an attribute has the same domain, I have introduced the simplification that applicability does not change. This precludes certain types of change, e.g. an employee who changes his job within the same organization, or the introduction of new regulations which prescribe new information to be stored (or previously stored information not
to be stored). In these types of changes, we may find that some objects have not changed their identity but will have new attributes applicable to them. This amounts to replacing an \( f_1 \in a \) by an \( f_2 \in a \) with a different domain, which is impossible according to the above definition. The reason for introducing this simplification is that these more complex types of change cannot be described adequately before we have a formalism powerful enough to describe the simpler types of change.

3.1.3. The set \( \Sigma_A \) contains the **attribute axioms** or **aggregation axioms**:

1. \( NA \in S \).
2. \( a = [k_1 \rightarrow k_2] \) for each attribute name \( a \).
3. \( \forall a \in A (\exists k \in S. a = a_1 \land \ldots \land a = a_n) \).

Axiom 3 is a kind of closure axiom for attribute names. It is assumed that there are finitely many attributes. It may in some cases be more convenient to assume that there are infinitely many attributes in the domain. The database stores only finitely many of them and may compute attributes which are related to the stored attributes via static integrity constraints. If we do not know how many attributes will become relevant for the UoD, we do not know how many attributes are in \( A \). Treating \( A \) as an infinite set then represents the fact that we may at any time want to define, store or compute a new attribute.

In case \( A \) is infinite, axiom 3 becomes infinite. We can circumvent this by defining the set

\[ \mathcal{A}(S) = \{ [k_1 \rightarrow k_2] \mid k_1, k_2 \subseteq S \} \]

and replacing axioms 2 and 3 by

\[ \exists a : \infty \rightarrow \mathcal{A}(S). \]

\( a(i) \) can then be treated as a name for an attribute, just as we did with \( s_i \). However, this has the drawback that we cannot talk of the age of a person but must use \( a_i \) instead, where \( a_i \) is system-generated just as \( s_i \) is system-generated. For this reason, I assume that \( A \) is finite but large enough to contain all interesting attributes.\(^\text{11}\)

3.2. The example domain

---
\(^{11}\) The domain language has therefore limited discriminating powers, for two entities may agree in properties represented by attributes in \( A \) but not agree in the properties represented by attributes outside \( A \). As far as we can tell in the domain language, the states of those two entities are indistinguishable. Were it not for the use of (ad hoc) identities, the principle of Leibniz that indistinguishable entities are identical would not be satisfied. It is debatable whether the identity of an object is a property of that object. If it is not, Leibniz' principle is violated.
3.2.1.
I use the ubiquitous and simple project/employee example in this report. All employees are
described by an employee number and a name, and projects by project number, project type,
and their set of members. Each project member has a task in the project. An employee can be
a member of arbitrary many projects.

The set of employees can be partitioned into truck drivers, secretaries and engineers or
alternatively into temporary and permanent employees. Truck drivers have a drivers licence,
secretaries a typing speed, engineers a skill, temporary employees a contract, and permanent
employees a period in which they are in service. A particular type of truck drivers is the type
of long-distance truck drivers, which in addition to the usual drivers licence have an interna­tional drivers licence. Some structures will be added for illustrative purposes later on.

3.2.2.
To write down the attribute axioms, I need some names for defined sets. Two defined sets I use
without further comment are $\emptyset$ and $\mathbb{N}$. All other defined sets are written in capital letters.
One such set is $\text{ALPHA}$, which may be thought of as the set of alphabetical entities. The nature
of the elements of $\text{ALPHA}$ is irrelevant for the formal theory, but we may imagine that their ele­ments
will be projected on the screen of a terminal in digestible form as letters and other sym­bols. The function $\text{STRINGS}(x)$ assigns to $x$ the set of all tuples of arbitrary length of elements
of $x$ and is defined in appendix 2. $\text{NAMES}$ is defined as $\text{STRING}(\text{ALPHA})$. Definitions of the
other defined sets are omitted.

Operations may be defined for the defined sets (such as concatenation for $\text{NAMES}$), so
that they may be thought of as abstract data types.

3.2.3.
The attribute axioms for our example are

$$\Sigma_A = \{ NA \in S \},$$

\begin{align*}
\forall a \in A [a = \text{emp#} \lor \ldots \lor a = \text{task}], \\
\text{ext} = [\text{Ext} \rightarrow \hat{\mathcal{S}}(S)], \\
\text{emp#} = [\text{Emp} \rightarrow^{-1} \text{EMP##}], \\
\text{salary} = [\text{Emp} \rightarrow \mathbb{N}^+] , \\
\text{name} = \text{Emp} \cup \text{Project} \rightarrow \text{NAMES}, \\
\text{licence#} = [\text{Trucker} \rightarrow^{-1} \text{LICENCE##}], \\
\text{typing-speed} = [\text{Secr} \rightarrow \text{SPEEDS}], \\
\text{skill} = [\text{Eng} \rightarrow \text{SKILLS}], \\
\text{contr#} = [\text{Contract} \rightarrow^{-1} \text{CONTRACT##}], \\
\#\text{years} = [\text{Perm} \rightarrow \mathbb{N} ], \\
\text{project#} = [\text{Project} \rightarrow^{-1} \text{PROJECT##}], \\
\text{project-type} = [\text{Project} \rightarrow \text{PROJECT-TYPES}], \\
\text{members} = [\text{Project} \rightarrow \mathcal{S}(\text{Emp} \times \text{TASK})].
\end{align*}
$\mathcal{F}(x)$ is the set of finite subsets of $x$.

3.2.4.

The attribute axioms define an aggregation graph as follows. The graph contains one node for each domain and range of an attribute and one directed edge for each attribute. The nodes are labelled by the expressions defining the domain and range of attributes. There is an arrow labelled $a$ from node labelled $k_1$ to the node $k_2$ if $a = [k_1 \to k_2]$. In general, the graph is disconnected, as shown in figure 3.

![Aggregation Graph](image)

Figure 3.

Note that we may have
Only if this is known not to be the case in the domain, can the aggregation graph be called an aggregation hierarchy. The domain can then be viewed as a collection of atoms (surrogates to which no attributes apply) aggregated into more complex objects. In general, there will be cycles in the aggregation graph. An example in a UoD of employees, managers and departments is

\[
\begin{align*}
\text{manager} &= \text{[Dept} \rightarrow \text{Emp]} \\
\text{dept} &= \text{[Emp} \rightarrow \text{Dept]}. 
\end{align*}
\]

### 3.3. Attributes, properties and operations

#### 3.3.1. Definition

For \( f \in a \) and \( x \in S \), \( f(x) \) is called an attribute value or contingent property of \( x \). A function \( f : S \times \cdots \times S \rightarrow S \) is called an operation. \( f(x_1, \ldots, x_n) \) is called a necessary property of \( x_1, \ldots, x_n \).

Later, the notion of property will be related to worlds (domain states). For a possible world \( w \), I will define the attribute value of \( a \) on \( x \) in \( w \) as \( a_w(x) \). This value is a contingent property of \( x \) in \( w \). \( x \) may have different contingent properties for the same attribute in different worlds. For example, a car may be red in one world but blue in another. In these two possible states, the car has two different contingent properties.

This contrasts with operations. For example, a set \( y \) of cars has a cardinality, which is the value of the unary operation \( |\cdot| \) on \( x \). The cardinality of \( y \) is the same in all possible worlds. In terms of modal logic, it is a necessary property of \( y \) that it has the cardinality it has; it could not be otherwise. Thus, the color of a car is a contingent property of that car, but the cardinality of a set of cars is a necessary property of that set.

#### 3.3.2.

An attribute is called variable if its value may be changed. Otherwise it is called constant.

For example, the color of a car is a variable attribute. In a domain where people may not change names, \( \text{name} \) is a constant attribute of a person. Note that the \( \text{name} \) of a person is contingent (it could have been otherwise) but constant (once given, a name must not be changed). These definitions can be given more precisely when a state transition function has been defined for objects. This is done in Wieringa [forthcoming], but because the distinctions made are so important already in the static phase of domain specification, I draw them here already.

---

12. That this must be the case, as Leibniz (and the early Wittgenstein) thought, seems to be a matter of metaphysical belief for which no good argument has yet been found. "And there must be simple substances; for the composite is nothing else than an accumulation or aggregate of the simple ... Thus these monads are the veritable atoms of nature, and in one word, the elements of all things." Paragraphs 2 and 3 of the *Monadology*, in P. Schrecker & A.M. Schrecker (trans.), G.W. Leibniz, *Monadology and Other Essays*, Un. of Stockholm, 1978.
Chapter 4  
The generalization hierarchy

4.1. Types and kinds, natural and unnatural

4.1.1. There are a number of terms in use to indicate the nature of a collection of objects. In computer science, the words "type," "class," "sort," and "abstract datatype" are in use. In mathematics, "set," "class" (in a different meaning) and "bag" are common. Looking further abroad, in philosophy the term "natural kind" is used to indicate a collection of objects which are similar in structure. In biology, terms like "species," genus," "order," and "family" are used to describe the classes into which living things can be ordered. I use some of these terms with meanings which at once stay close to what is usually intended with them in the sciences they are borrowed from and given precise meaning in the ABSURD model. A preliminary listing follows.

1. **Natural kind.** This will be used for a set of surrogates with the same aggregation structure, i.e. the set of all possible object identities such that the same attributes are applicable to the identities. In relational terms, this corresponds to the intension of a relation scheme. The term "natural kind" is inspired by linguistic philosophy, where it means a set of objects with the same underlying (i.e. empirically validated, scientific) structure (see Schwartz [1977], especially Quine's essay). A much used example of a natural kind is gold. Natural kinds have names that are **rigid designators,** i.e. the name refers to the same set of possible objects in all possible worlds.

   To the idea of structural isomorphy is added the dynamic concept of dynamic isomorphy. In Wieringa [forthcoming], all members of a natural kind execute the same general process. Thus, employees, projects, departments etc. have the same static and dynamic structure.

   This does not exhaust the concept of natural kind in DB domain specification, but it goes a long way. For example, all persons form a natural kind because they are isomorphic (at a certain level of abstraction) and have the same life cycle. But this isomorphy is just isomorphy in attribute applicability. That excludes specialization to subkinds by restriction on allowable attribute values. People below 65 years have the same aggregation structure as people above 65 years of age and also execute the same generic process. But they may be said to belong to different natural subkinds of the natural kind People, for they enjoy different privileges (in Holland) concerning public transport, cultural activities etc. The problem is, not any arbitrary predicate on attribute values selects a natural kind. The set of people above 67 years of age whose surname begins with an "S" is not a natural kind -unless there is a generic process applicable to precisely that set of people. This suggests that the concept could be extended from the conjunction "same aggregation structure and same generic process" to the disjunction "same aggregation structure or same generic process."

   But even this does not exhaust the concept of natural kind. In biology, the concept of
natural kinds crucially involves common descent (Mayr [1982]). This may be relevant to DB domains in as far as there, natural kinds evolve as well. New attributes emerge and old ones become obsolete. But again, not just any replacement of an aggregation structure or generic process by any other will be acceptable as evolution of natural kinds. Investigation of these problems will be a topic for future research.

2. **Kind.** Any set of surrogates is called a kind. This is just to have a uniform terminology. Some kinds have the special property of being natural.

3. **Type.** Any set of attributes is called a type. This corresponds to the relation scheme of relational theory. If a type is the type of a natural kind, it is called a natural type.

4. **Class.** Any set of objects is called a class. If the identities of the objects in a class are a natural kind, we call it a natural class. The elements of a class are thus objects with an internal state. This agrees with the concept of class in object-oriented systems. We can have two classes consisting of objects with the same set of identities but in different states. Those classes are then of the same kind.

5. **Extension.** A kind is a set of possible object identities. I define a world as a set of objects with different identities and call the set of identities of objects in a world the existence set of that world. The intersection of a kind $k$ with the existence set of $w$ is called the extension of $k$ in $w$. This corresponds roughly to the relation instance of relational theory. We can also talk of the extension of a type or of a class in a world.

There are a number of related concepts not mentioned in the above list. Most prominently is the concept of Abstract Data Type (ADT), which, according to one definition, is a collection of sets (called sorts) together with functions on those sets (called operations). ADT’s are timeless versions of natural kinds. In Wieringa [forthcoming] I define for each natural kind a set of events and a generic process composed of those events. Each object of the natural kind executes an instance of the generic process of the kind. The natural kind corresponds to the sort of an ADT and the set of events to the operations defined on the sort. If the natural kind has sub-kinds, these correspond to subsorts of a sort in a many-sorted ADT. The difference is that

1. the objects of a natural kind have an internal state and those of an ADT do not, and
2. the effect of an event depends upon the state of the domain in which it is executed (it is contingent) whereas the result of an operation is independent of the state of the world (it necessarily gives the same result in any world).

ADT’s appear in this report as atomic kinds (e.g. $\mathbb{N}$).

4.1.2. **Definition**

A subset of $A$ is called a type. The letters

$$t_0, t_1, \ldots,$$

will be used as metavariables over types.

The largest type of $k$ is the set

$$\text{type}(k) = \{ a \in A \mid k \subseteq \text{dom}(a) \}.$$  

Each subset of $\text{type}(k)$ is called a type of $k$. □
4.1.3. Definition
A subset of $S$ is called a kind. We use the letters $k_0, k_1, \ldots$ as metavariables over kinds. If $t$ is a type, the kind of $t$ is the set

$$\text{kind}(t) = \{x \in S \mid \forall a \in t x \in \text{dom}(a)\}.$$

Thus, any set of attributes is a type and any set of identities a kind. We can prove a number of general things for kinds and types. In the following theorem, $t$ and $k$ are universally quantified. Thus, 4 is an abbreviation of

$$\forall k \subseteq S \forall t \subseteq A [k \subseteq \text{kind}(t) \Rightarrow t \subseteq \text{type}(k)].$$

4.1.4. Theorem
1. $\text{kind}(\emptyset) = S$.
2. $\text{type}(\emptyset) = A$.
3. $\text{kind}(t) = \bigcap_a \text{dom}(a)$.
4. $k \subseteq \text{kind}(t) \Rightarrow t \subseteq \text{type}(k)$.
5. $t_1 \subseteq t_2 \Rightarrow \text{kind}(t_2) \subseteq \text{kind}(t_1)$.
6. $k_1 \subseteq k_2 \Rightarrow \text{type}(k_2) \subseteq \text{type}(k_1)$.

Proof.
1 and 2. Immediate.
3. $x \in \text{kind}(t) \Rightarrow \forall a \in t \forall x \in \bigcap_a \text{dom}(a)$.
4. $k \subseteq \text{kind}(t) \Rightarrow \forall x \in k \forall a \in t \forall x \in \bigcap_a \text{dom}(a)$.
5. If $\text{kind}(t_2) = \emptyset$, then $\text{kind}(t_2) \subseteq \text{kind}(t_1)$. So let $\text{kind}(t_2) \neq \emptyset$ and choose $x \in \text{kind}(t_2)$. Then $x \in \text{kind}(t_2) \Rightarrow \forall a \in t_2 x \in \text{dom}(a)$. But $t_1 \subseteq t_2$, so $\forall a \in t_1 x \in \text{dom}(a)$, so $x \in \text{kind}(t_1)$.
6. If $k_1 \subseteq k_2$, $\forall a \in \text{type}(k_2) \Rightarrow k_2 \subseteq \text{dom}(a)$.

The duals of 1 and 2 are not true in general. $\text{type}(S)$ is the set of attributes applicable to all identities, so $\text{type}(S) = \emptyset$ if there is no attribute applicable to all identities.

$\text{kind}(A)$ is the set of identities to which all attributes are applicable, so $\text{kind}(A) = \emptyset$ if there is no identity to which all attributes apply. But by definition, we have $\emptyset \in \text{kind}(A)$.

Part 3 says simply that the kind of a type is just the intersection of the domains of the attributes in the type. The intuitive meaning of 4 is that if we remove an attribute from a type, we may increase the set of identities to which the remaining attributes are applicable. Similarly, if we remove an identity from a kind, we may get a kind to which more attributes are applicable.

It is not in general true that
kind \( (t) \subseteq k \iff \text{type} (k) \subseteq t \).

As an example of

\[ \text{kind} (t) \subseteq k \iff \text{type} (k) \subseteq t, \]

take the kind \( \{ \text{emp}\# , \text{dept}\# \} \). Using intuitive interpretations of the attribute names (employee number and department number), \( \text{kind} (\{ \text{emp}\# , \text{dept}\# \}) = \emptyset \nsubseteq k \) for any \( k \subseteq S \). But in general, for any \( k \subseteq S \), \( \text{type} (k) \nsubseteq \emptyset \).

As an example of

\[ \text{type} (k) \subseteq t \iff \text{kind} (t), \]
take \( \text{John} \) and \( d \) to be names of identities for a particular person and a particular department in a domain where persons and departments have nothing in common. Then \( \text{type} (\{ \text{John} , d \}) = \emptyset \nsubseteq t \) for any \( t \). But in general, \( \text{kind} (t) \nsubseteq \{ \text{John} , d \} \).

Similarly, to show that the converse of 6 does not hold in general, we have \( \text{type} (\{ \text{John} , d \}) \nsubseteq \text{type} (k) \) for any \( k \subseteq S \), but in general, \( k \nsubseteq \{ \text{John} , d \} \).

Finally, as a counterexample of the converse of 5, let \( \text{kind} (\{ \text{typing-speed} \}) \) be the set of secretaries and \( \text{kind} (\{ \text{contract}\# \}) \) be the set of temporary employees. If management has decreed that only secretaries can be temporary employees, then \( \text{kind} (\{ \text{contract}\# \}) \nsubseteq \text{kind} (\{ \text{typing-speed} \}) \). But \( \{ \text{typing-speed} \} \nsubseteq \{ \text{contract}\# \} \).

The following two simple corollaries of the theorem introduce the definition of natural kinds (types). \( t \) and \( k \) are again universally quantified.

4.1.5. Corollary

1. \( t \subseteq \text{type} (\text{kind} (t)) \).
2. \( k \subseteq \text{kind} (\text{type} (k)) \).

Proof.

1. By 4.1.4.4, \( \text{kind} (t) \nsubseteq \text{kind} (t) \iff t \subseteq \text{type} (\text{kind} (t)) \).
2. Analogous. \( \Box \)

If we take a set \( k \) of identities, gather the set \( t \) of attributes applicable to all members of \( k \), then \( k \) will be contained in the set of all identities to which the attributes in \( t \) are applicable. This strongly suggests the following definition for natural kinds.

4.1.6. Definition

A natural kind is a set of identities such that \( k = \text{kind} (\text{type} (k)) \). A natural type is a type such that \( t = \text{type} (\text{kind} (t)) \). We use

\[ k_0 , k_1 , \ldots , \] as metavariable over the natural kinds and
\[ t_0 , t_1 , \ldots , \] as metavariable over natural types. \( \Box \)

There is a simple criterion for natural kindhood, stated in the following theorem.
4.1.7. Theorem

1. \( k \) is a natural kind iff there is a type \( t \) with \( k = \text{kind}(t) \).
2. \( t \) is a natural type iff there is a kind \( k \) with \( t = \text{type}(k) \).

Proof.

1. If \( k \) is a natural kind, then there is a \( t \) with \( k = \text{kind}(t) \), for take \( t = \text{type}(k) \). Conversely, assume that \( k = \text{kind}(t) \) for a \( t \). Then by 4.1.5.1, \( t \subseteq \text{type}(\text{kind}(t)) \), so that
   \[
   \text{kind}(\text{type}(\text{kind}(t))) \subseteq \text{kind}(t).
   \]
   But by 4.1.5.2,
   \[
   \text{kind}(t) \subseteq \text{kind}(\text{type}(\text{kind}(t))),
   \]
   so \( \text{kind}(t) = \text{kind}(\text{type}(\text{kind}(t))) \), so that \( k \) is a natural kind.
2. Analogous. \( \square \)

A natural kind is thus the largest kind of a type. For natural kinds and types, the counterexamples given above are not valid and we have the following theorem. (All variables universally quantified.)

4.1.8. Theorem

1. \( \text{type}(\emptyset) = \emptyset \Rightarrow \emptyset = S \).
2. \( \text{kind}(\emptyset) = \emptyset \Rightarrow \emptyset = A \).
3. \( \text{kind}(\emptyset) \subseteq \emptyset \Rightarrow \text{type}(\emptyset) \subseteq \emptyset \).
4. \( \emptyset \subseteq \emptyset \Rightarrow \text{type}(\emptyset) \subseteq \text{type}(\emptyset) \).
5. \( \emptyset \subseteq \emptyset \Rightarrow \text{kind}(\emptyset) \subseteq \text{kind}(\emptyset) \).

Proof.

1. \( \emptyset = \text{kind}(\text{type}(\emptyset)) = \text{kind}(\emptyset) = S \).
2. Analogous.
3. \( \text{kind}(\emptyset) \subseteq \emptyset \Rightarrow \text{type}(\emptyset) \subseteq \text{type}(\emptyset) \Rightarrow \text{type}(\emptyset) \subseteq \emptyset \).
4. \( \emptyset \subseteq \emptyset \Rightarrow \text{kind}(\emptyset) \subseteq \text{kind}(\emptyset) \Rightarrow \emptyset \subseteq \text{kind}(\emptyset) \).
5. Analogous. \( \square \)

4.1.9. Corollary

1. \( \text{kind}(\emptyset) = \emptyset \Rightarrow \emptyset = \text{type}(\emptyset) \).
2. \( k \) is a natural kind iff \( \text{type}(k) \) is a natural type.
3. \( t \) is a natural type iff \( \text{kind}(t) \) is a natural kind.

Proof.

1. Immediate.
2. \( k \) is a natural kind iff \( k = \text{kind}(\text{type}(k)) \) iff \( \text{type}(k) = \text{type}(\text{kind}(\text{type}(k))) \) iff \( \text{type}(k) \) is a natural type.
3. Analogous. \( \square \)
For natural kinds and types, \textit{kind} and \textit{type} are each other's inverse. The type of a natural kind is natural and vice versa. All of this suggests that the natural kind lattice is inverted in the natural type lattice. The next section shows that this is not quite so.

The following theorem gives another criterion for natural kinds and types.

4.1.10. Theorem

1. \( k \) is a natural kind iff \( \forall k' [\text{type}(k) = \text{type}(k') \Rightarrow k' \subseteq \text{dom}(k)] \).
2. \( t \) is a natural type iff \( \forall t' [\text{kind}(t) = \text{kind}(t') \Rightarrow t' \subseteq t] \).

Proof.

1. If \( k \) is a natural kind, then \( \text{type}(k) = \text{type}(k') \Rightarrow \text{kind}(\text{type}(k)) = \text{kind}(\text{type}(k')) \Rightarrow k = \text{kind}(\text{type}(k')) \supseteq k' \).

Conversely, given \( \forall k' [\text{type}(k) = \text{type}(k') \Rightarrow k' \subseteq k] \) we have to prove \( \text{kind}(\text{type}(k)) \subseteq k \). This follows from \( \text{type}(\text{kind}(\text{type}(k))) = \text{type}(k) \) and the premiss, so we have to prove \( \text{type}(\text{kind}(\text{type}(k))) \subseteq \text{type}(k) \). If the left hand side is empty, there is nothing to prove, so choose \( a \in \text{type}(\text{kind}(\text{type}(k))) \). Then \( \text{kind}(\text{type}(k)) \subseteq \text{dom}(a) \), so \( k \subseteq \text{dom}(a) \), so \( a \in \text{type}(k) \).

2. Analogous. \( \square \)

A set of two truck drivers and a set of three truck drivers are both of the same type. This theorem says that that natural kind of truck drivers is the largest of the sets of that type, which is the set of all actual and possible truck drivers.

The set of all actual and possible truck drivers and one secretary is not a natural kind, because we can add more secretaries without changing type. The largest set of objects which have that same type is a natural kind again. This may well be the set of employees.

Continuing in this vein, we get the largest possible natural kind, which is \( S \).

Similarly, a natural type is the largest set of attributes of a given kind. In our example, \{\text{emp\#}\} is not a natural type because \text{name} is applicable to employees as well. \{\text{emp\#, name}\} is a natural type, for adding one more attribute would decrease the kind of the type. \{\text{emp\#, project\#}\} is not a natural type, for \{\text{emp\#, project\#, name}\} describes the same kind, viz. \( \emptyset \). The largest natural type is \( A \).

4.1.11. Corollary

1. \( S \) is a natural kind and \( A \) is a natural type.
2. \( k_0 \) is a natural kind, where \( k_0 \) is the largest set of identities to which all attributes apply.
3. \( t_0 \) is a natural type, where \( t_0 \) is the largest set of attributes applicable to all identities.

Proof.

Immediate. \( \square \)

Usually, \( k_0 = t_0 = \emptyset \).

The concept of natural kind introduced here is minimal in that it is just the largest set of (identities of) objects described by a type. Subkinds which are described by the same attributes but have different attribute values are not counted as natural kinds in this view, unless an extra
attribute is applicable to identities of the subkind.

For example, the set of employees is a natural kind and the set of employees and one screw-driver is not. However, the set of senior employees will be a natural kind according to our criterion only if it is the largest set to which a given set of attributes applies. If there is no extra attribute applicable to senior employees, our criterion will not single it out as a natural kind. The criterion of attribute applicability is thus not comprehensive, but I claim that at least it does not elevate unnatural kinds to natural kindhood.

4.1.12.

The set $CON_{NK}$ contains the constant symbol $NK$ and names for natural kinds in the domain. we use

\[ \ell_0, \ell_1, \ldots, \]

as metavariable over $CON_{NK}$. $\Sigma_{NK}$ is the smallest set of sentences containing the following axioms.

1. $NK \subseteq S$.
2. $\ell_i \in NK$ for each natural kind name in $\ell_i$ in $CON_{NK}$.
3. $\text{kind}(\text{type}(\ell_i)) = \ell_i$ for each natural kind name $\ell_i$ in $CON_{NK}$.

Because a type is natural iff its kind is natural, we do not introduce names for natural types. Note that we do not have a "natural kind name closure axiom." In the next section we prove that the intersection of two natural kinds is a natural kind, and having to think of names for all non-empty intersections of natural kinds is a bit tedious.

We use names which start with a capital letter for natural kinds. In our example, we introduce the following natural kind names and axioms.

\[
CON_{NK} = \{ S, \text{Emp}, \text{Trucker}, \text{Ldtrucker}, \text{Secr}, \text{Eng}, \text{Temp}, \text{Perm}, \text{Project} \}
\]

\[
\Sigma_{NK} = \{ NK \subseteq \emptyset(S) \\
S, \text{Emp}, \ldots, \text{Project} \subseteq NK, \\
\text{kind}(\text{type}(\text{Emp})) = \text{Emp}, \\
\text{kind}(\text{type}(\text{Project})) = \text{Project} \}.
\]

4.2. Attribute inheritance

4.2.1. Definition

If $t_1 \subseteq t_2$, $t_2$ is called a specialization of $t_1$ and $t_1$ a generalization of $t_2$. $t_2$ is said to inherit the attributes in $t_1$. We also say that $\text{kind}(t_1)$ is a specialization of $\text{kind}(t_2)$.

In other words, to specialize a type is to add attributes, to generalize it is to remove attributes. This is the most general type of specialization, specialization by attribute inheritance.

The following theorem shows why the kind and type lattices are not completely each other's inverse. There is a dual to this theorem which can be gotten by replacing kind by type.
and  by  

4.2.2. Theorem

1. \( \bigcap_{i=1}^{n} \text{kind}(t_i) = \text{kind}(\bigcup_{i=1}^{n} t_i) \).

2. \( \bigcup_{i=1}^{n} \text{kind}(t_i) \subseteq \text{kind}(\bigcap_{i=1}^{n} t_i) \)

Proof.

1. \( t_j \subseteq \bigcup_{i=1}^{n} t_i \) for \( j = 1, \ldots, n \), so \( \text{kind}(\bigcup_{i=1}^{n} t_i) \subseteq \text{kind}(t_j) \) for \( j = 1, \ldots, n \) and therefore \( \text{kind}(\bigcup_{i=1}^{n} t_i) \subseteq \bigcap_{i=1}^{n} \text{kind}(t_i) \). To prove the reverse, let \( x \in \bigcap_{i=1}^{n} \text{kind}(t_i) \). Then \( x \in \text{kind}(t_i) \) for \( i = 1, \ldots, n \), so \( \forall a \in t_i[x \in \text{dom}(a)] \) for \( i = 1, \ldots, n \) and therefore, \( \forall a \in \bigcup_{i=1}^{n} t_i[x \in \text{dom}(a)] \) so that \( x \in \text{kind}(\bigcup_{i=1}^{n} t_i) \).

2. \( \bigcap_{i=1}^{n} t_i \supseteq t_j \) for \( j = 1, \ldots, n \), so \( \text{kind}(t_j) \subseteq \text{kind}(\bigcap_{i=1}^{n} t_i) \), for \( j = 1, \ldots, n \) and therefore \( \bigcup_{i=1}^{n} \text{kind}(t_j) \subseteq \text{kind}(\bigcap_{i=1}^{n} t_i) \).

The reverse of 2 is not true in general, for

\( x \in \bigcup_{i=1}^{n} \text{kind}(t_i) \not\supset s \in \text{kind}(t_i), \ i = 1, \ldots, n. \)

Consider the types \( t_1, \ldots, t_n \). Their least upper bound (lub) is \( \bigcup_{i=1}^{n} t_i \) and their greatest lower bound (glb) is \( \bigcap_{i=1}^{n} t_i \). The theorem says that the lub of \( t_i \) goes with \( \text{kind} \) to the glb of the corresponding set of kinds, but that the glb of \( t_i \) does not necessarily go to the lub of the corresponding kinds. Figures 4 and 5 show what is going on.

Figure 4 shows multiple attribute inheritance. \( t \) inherits all attributes in each \( t_i \) and is mapped by \( \text{kind} \) to the intersection of the \( \text{kind}(t_i) \).

Figure 5 shows single inheritance. The \( t_i \) inherit all attributes from \( t \) and each possibly has some extra attributes as well. \( \text{kind}(t) \) contains all \( \text{kind}(t_i) \) but may contain extra identities as well. For example, if \( \{\text{emp#}, \text{name}\} \) is the type of all employees and \( \{\text{emp#}, \text{name}, \text{skill}\} \) and \( \{\text{emp#}, \text{name}, \text{typing-speed}\} \) are the types of engineers and secretaries, respectively, then
kind(\{emp\#, name\, skill\} \cup \{emp\#, name\, typing-speed\}) \subseteq kind(\{emp\#, name\}).

But in case there is also a type \{emp\#, name, licence\#\} of employees who are truck drivers, $kind(\{emp\#, name\})$ is not exhausted by the sets of secretaries and engineers. In general, when we specialize a type $t$ to types $t_i$, we may fail to be able to describe all identities in $kind(t)$ by $t_i$. 
To reach clarity in the domain structure (and to facilitate the definition of processes executed by members of a natural kind), we are interested in specializations which partition the kind of the generalization, i.e. where \( \text{kind}(t) = \bigcup \text{kind}(t_i) \) and the \( \text{kind}(t_i) \) are pairwise disjoint.

4.2.3. Definition

The kinds \( k_i, i = 1, \ldots, n \) partition \( k \) if \( k = \bigcup k_i \) and the \( k_i \) are pairwise disjoint. They are a subdivision of \( k \) if \( \bigcup k_i \subseteq k \) and the \( k_i \) are pairwise disjoint.

The types \( t_i, i = 1, \ldots, n \) are a specialization group of \( t \) if the \( \text{kind}(t_i) \) subdivide \( \text{kind}(t) \). They are an exhaustive specialization group if the subdivision is a partition.

An example of a kind with two partitions is the kind \( \text{emp} \) of employees, partitioned by secretaries, engineers and truck-drivers as well as by temporary-ems and permanent-ems. It must be emphasized that it may not be known to the system designer or user whether a set of kinds partitions or even subdivides a given kind. For example, a new kind of employee, department heads, may be hired, who differ in kind from any employee hired hitherto. Since \( \text{emp} \) contains all past, present and future actual and possible identities, the division by secretaries, engineers and truck-drivers has never been exhaustive, except that we did not know it. Similarly, we may create a kind of employees who double as secretaries and engineers. If we previously though these two kinds were disjoint, we were wrong.

The declaration that a kind is partitioned or subdivided by a group of kinds should be viewed as a belief about the domain at the time we make that statement; or alternatively, it may be viewed as a decision to constrain the behavior of the domain in a certain way, e.g. as the decision never to hire department heads; or again, as the decision not to represent the hiring of department heads.

We usually declare a group of kinds \( k_i \) to be a subdivision or partition of \( k \) by declaring \( \text{type}(k_i) \) to be an (exhaustive) specialization group of \( \text{type}(k) \). The following theorem explicates what it is we are then declaring.

4.2.4. Theorem

1. \( \text{kind}(t_i \cup t_j) = \emptyset, i \neq j \) iff the \( t_i \) are a specialization group of \( \cap t_i \).
2. \( \text{kind}(\cap t_i) \subseteq \bigcup \text{kind}(t_i) \Leftrightarrow \text{kind}(\cap t_i) = \bigcup \text{kind}(t_i) \).

Proof.

Trivial. \( \square \)

We do not want to give subdivision or partition relations between any odd group of kinds, but only between natural kinds. For natural kinds, the first part of the previous theorem can be simplified a bit.

4.2.5. Theorem

If \( t, t_i \) are the natural types corresponding to \( k, k_i, i = 1, \ldots, n \), then

1. \( t \subseteq \cap t_i \Leftrightarrow \bigcup k_i \subseteq k \).
2. \( t = \cap t_i \Leftrightarrow \bigcup k_i = k \).
Proof.
Trivial. □

When we provide names for natural kinds in $CON_{NK}$, it is good to know that $NK$ is a meet-semilattice (and similarly, the set of natural types is a meet-semilattice).

4.2.6. Theorem
If $k_1 \cap k_2 \neq \emptyset$, then it is a natural kind.

Proof.
Let $type(k) = type(k_1 \cap k_2)$, then to prove that $k \subseteq k_1 \cap k_2$. Of $k = e.pry$, there is nothing to prove, so choose $x \in k$. Then

\[
\{x\} \subseteq k \Rightarrow \\
type(k) \subseteq type(\{x\}) \Rightarrow \\
type(k_1 \cap k_2) \subseteq type(\{x\}) \Rightarrow \\
type(k_1 \cap k_2) \cap type(k_1) \subseteq type(\{x\}) \cap type(k_1) \Rightarrow \\
type(k_1) \subseteq type(\{x\} \cap k_1).
\]

But also $k_1 \subseteq k_1 \cup \{x\} \Rightarrow type(\{x\} \cup k_1) \subseteq type(k_1)$, so $type(k) = type(k_1 \cup \{x\})$. Because $k \in NK$, $k_1 \cup \{x\} \subseteq k_1$, so $x \in k_1$. So $k \subseteq k_1$ and by symmetry, $k \subseteq k_2$. So $k \subseteq k_1 \cap k_2$. □

It is not true in general that $k_1 \cup k_2 \in NK$. For example, let $Emp$ be the natural kind of employees and $Dep$ the natural kind of departments. If $Proj$ is the set of projects, then $type(Emp \cup Dep) = \emptyset = type(Proj \cup Dep)$, but $Dep \cup Proj \subseteq Emp \cup Dep$.

4.2.7. Definition
The meet-semilattice formed by $NK$ is called the specialization hierarchy of the domain. □

We like the specialization hierarchy to be as well-structured as possible, i.e. to allow only partitions and subdivisions. These are defined by a set $\Sigma_{sp}$ of specialization axioms for natural kinds. For each natural kind name $k$ in $CON_{NK}$ and subdivision $k_i$ of $k$ we add the sentences

1. $\cup k_i \subseteq k$
2. $k_i \cap k_j$ for $i \neq j$


to $\Sigma_{sp}$. Similarly for partitions, where in 1 we have an equals sign.

The specialization axioms in our example are

\[
\Sigma_{sp} = \{S = Emp \cup Project \cup Project - member, \\
Emp \cap Project = \emptyset, Emp \cap Project - member = \emptyset, Project \cap Project - member = \emptyset, \\
Emp = Trucker \cup Secr \cup Eng, \\
Trucker \cap Secr = \emptyset, Trucker \cap Eng = \emptyset, Secr \cap Eng = \emptyset, \\
Labtruckers \subseteq Trucker, \\
Emp = Temp \cup Perm, \\
Temp \cap Perm = \emptyset\}.
\]

Figure 6 visualizes these axioms as a directed graph (the implicit direction of the edges is downwards). An arc through edges leaving a node indicates pairwise disjointness of the nodes pointed at.
Note that, by theorem 4.2.6, the following kinds are natural in addition to the ones shown in the figure:

- Trucker ∩ Temp, Trucker ∩ Perm,
- Secr ∩ Temp, Secr ∩ Perm,
- Eng ∩ Temp, Eng ∩ Perm,
- Ldtrucker ∩ Temp, Ldtrucker ∩ Perm.

4.2.8.

We now have introduced all axioms of $\Sigma_D$. In appendix 4, a model for $\Sigma_D$ is exhibited. Basically, the idea is to interpret $S$ as $\mathbb{N}$ and define the natural kinds in such a way that $\Sigma_{SP}$ is satisfied. This also gives an idea how a model for a domain theory can be implemented, by assigning global database identifiers to objects in such a way that the form of the identifier betrays the kind of the object identified.

4.3. Classifications

4.3.1. Definition

The classifications of $x \in S$ are the set

$$\text{classes}(x) = \{ k \mid \exists t \subseteq A [ k = \text{kind}(t) \land t \subseteq \text{type}(\{x\})] \}. \quad \Box$$

The classifications of an identity are the set of all natural kinds as which it can be classified.
The largest set of attributes applicable to \( x \) is \( \text{type}(\{x\}) \), so that the smallest natural kind of \( x \) is \( \text{kind}(\text{type}(\{x\})) \). We prove these statements and some other simple properties of \( \text{classes}(x) \) below.

### 4.3.2. Theorem

Let \( x \in S \).

1. If \( k \in \text{classes}(x) \), then \( x \in k \).
2. \( \text{lub}(\text{classes}(x)) = S \in \text{classes}(x) \).
3. \( \text{glb}(\text{classes}(x)) = \text{kind}(\text{type}(\{x\})) \in \text{classes}(x) \).
4. If \( k_1, \ldots, k_n \in \text{classes}(x) \), then \( \bigcap_{i=1}^n k_i \in \text{classes}(x) \).

**Proof.**

1. Trivial.
2. \( \text{lub}(\text{classes}(x)) = \bigcup_{t \subseteq \text{type}(\{x\})} \text{kind}(t) = S \in \text{classes}(x) \), because \( \emptyset \subseteq \text{type}(\{x\}) \) and \( \text{kind}(\emptyset) = S \).
3. \( \text{glb}(\text{classes}(x)) = \bigcap_{t \subseteq \text{type}(\{x\})} \text{kind}(t) = \text{kind}(\bigcup_{t \subseteq \text{type}(\{x\})} t) = \text{kind}(\text{type}(\{x\})) \in \text{classes}(x) \).
4. Because \( k_i \in \text{classes}(x) \), \( k_i = \text{kind}(t_i) \) for a \( t_i \subseteq \text{type}(\{x\}) \). But \( \bigcap_{i=1}^n t_i = \text{kind}(\bigcap_{i=1}^n t_i) = \text{kind}(t_0) \), where \( t_0 = \bigcap_{i=1}^n t_i \). But because \( t_i \subseteq \text{type}(\{x\}) \), \( t_0 \subseteq \text{type}(\{x\}) \), and so \( \bigcap_{i=1}^n k_i = \text{kind}(t_0) \in \text{classes}(x) \).

Part 2 says that \( S \) is the largest kind of any identity. It is the weakest classification of \( x \) and says nothing else that the identity possibly exists. Moreover, \( S \) is the union of all the classes of \( x \). Part 3 says that the smallest kind of a identity is the kind of its largest type, \( \text{type}(\{x\}) \), and part 3 and 5 imply that this is the intersection of all the natural kinds of \( x \). This proves the following corollary.

### 4.3.3. Corollary

1. \( \text{kind}(\text{type}(\{x\})) = \bigcap_{k \in \text{classes}(x)} k \).
2. \( S = \bigcup_{k \in \text{classes}(x)} k \).

**Proof.** Immediate. □

For example, let \( l_0 \) be a long distance truck driver in permanent employment. Then

\[
\text{classes}(l_0) = \{ \text{Ldtrucker, Ldtrucker } \cap \text{Perm, Trucker, Trucker } \cap \text{Perm, Emp, Emp } \cap \text{Perm, Perm, S} \}.
\]

These are the nodes on the paths in the specialization hierarchy from its smallest kind, \( \text{Ldtrucker } \cap \text{Perm} \), to its largest kind, \( S \). \( \text{Ldtrucker } \cap \text{Perm} \) is the bottom of the classifications of \( l_0 \). We can prove some properties of the bottom of the taxonomic hierarchy for a identity.
4.3.4. Definition
The smallest kind of \( x \in s \) is
\[
\text{classes}(x) = \text{kind}(\text{type}(\{x\})).
\]

To prove some properties of \( \text{classes}(x) \), the following lemma is useful.

4.3.5. Lemma
\[ \forall x, y \in S[ x \in \text{classes}(y) \Rightarrow \text{type}(\{y\}) \subseteq \text{type}(\{x\})]. \]

Proof.
For \( x, y \in S \) we have
\[ x \in \text{classes}(y) \Rightarrow x \in \text{kind}(\text{type}(\{y\})) \Rightarrow \text{type}(\{y\}) \subseteq \text{type}(\{x\}). \]

4.3.6. Theorem
1. \( \forall x \in S[ x \in \text{classes}(x) ] \).
2. \( \forall x, y, z \in S[ x \in \text{classes}(y) \land y \in \text{classes}(z) \Rightarrow x \in \text{classes}(z) ] \).
3. \( \forall x, y \in S[ y \in \text{classes}(x) \Rightarrow \text{classes}(y) \subseteq \text{classes}(x) ] \).
4. \( \forall x, y \in S[ y \in \text{classes}(x) \Rightarrow \text{classes}(x) \subseteq \text{classes}(y) ] \).

Proof.
1. By 4.3.2.1 and 4.3.2.3.
2. For \( x, y, z \in S \)
\[
x \in \text{classes}(y) \land y \in \text{classes}(z) \Rightarrow \\
\text{type}(\{y\}) \subseteq \text{type}(\{x\}) \land \text{type}(\{z\}) \subseteq \text{type}(\{y\}) \Rightarrow \\
\text{type}(\{z\}) \subseteq \text{type}(\{x\}) \Rightarrow \\
x \in \text{classes}(z).
\]
3. \[ \Rightarrow : \text{If } y \in \text{classes}(x) \text{ then by 2 for all } z \in \text{classes}(y) \text{ we have } z \in \text{classes}(x), \text{ so } \text{classes}(y) \subseteq \text{classes}(x). \]
\[ \Leftarrow : \text{Conversely, if } \text{classes}(y) \subseteq \text{classes}(x), \text{ then because } y \in \text{classes}(y), \text{ y } \in \text{classes}(x). \]
4. \[ \Rightarrow : \text{Let } y \in \text{classes}(x) \text{ and choose } k \in \text{classes}(x). \text{ Then there is a } t \subseteq \text{type}(\{x\}) \text{ with } k = \text{kind}(t). \text{ Because } \text{classes}(x) \text{ is the glb of } \text{classes}(x), \text{classes}(x) \supseteq k, \text{ so } y \in \text{kind}(t) \text{ and therefore } \{y\} \subseteq \text{kind}(t). \text{ But then } t \subseteq \text{type}(\{y\}), \text{ so } k = \text{kind}(t) \in \text{classes}(y). \]
\[ \Leftarrow : \text{Let } \text{classes}(x) \subseteq \text{classes}(y). \text{ Since } \text{classes}(x) \in \text{classes}(x), \text{classes}(x) \in \text{classes}(y). \text{ But then } \text{classes}(y) = \text{glb}(\text{classes}(y)) \subseteq \text{classes}(x). \text{ Because } y \in \text{classes}(y), \text{ y } \in \text{classes}(x). \]

Part 1 and 2 say that the relation \( x \in \text{skmind}(y) \) is reflexive and transitive, but 3 says that it falls short of being an equivalence relation. For example, if \( t_0 \in \text{Trucker} - \text{Ldirrucker} \) and \( t_0 \in \text{Trucker} - \text{Ldirrucker} \) and both are in permanent employment, then
\[ \text{classes}_\perp(l_0) = L\text{drucker} \cap \text{Perm} \text{ and} \]
\[ \text{classes}_\perp(t_0) = \text{Trucker} \cap \text{Perm}. \]

These two kinds are neither equal nor disjoint, since the first is genuinely contained in the second. Part 4 says that all natural kinds of \( t_0 \) are also natural kinds of \( l_0 \). As a matter of fact, \( l_0 \) has two more natural kinds than \( t_0 \), \( L\text{drucker} \) and \( L\text{drucker} \cap \text{Perm} \).

The situation would have been different if there would have been a kind \( S\text{drucker} \) (for short distance truck driver) such that \( L\text{drucker} \) and \( S\text{drucker} \) jointly partition \( \text{Trucker} \). Then all specialization groups would have been exhaustive and \( S \) would have been partitioned by the smallest kinds, so that \( x \in \text{classes}_\perp(y) \) would have been an equivalence relation.
Chapter 5
Objects and possible worlds

5.1. Objects

5.1.1.
Up till now, we have looked merely at the identities of objects. We now introduce the concept of state so as to define an object as a pair (identity, state). To define a state as a tuple of attribute values, we must associate a position in a state vector with an attribute. This is done simply by numbering the attributes.

5.1.2. Definition
A numbering of attributes is a bijection
$$\nu: A \rightarrow \mathbb{N} \text{ onto } |A|.$$ where $|.|$ denotes the cardinality of a set. □
For finite sets of attributes, the set $|A|$ is $\{0, \ldots, |A|-1\}$. The numbering $\nu$ is kept fixed during the rest of this paper.

5.1.3. Definition
For each finite $t \subseteq A$, define $\nu_t: t \rightarrow |t|$ by
$$\nu_t(a) = 0 \quad \text{iff} \quad \forall b \in t, \nu(a) \leq \nu(b)$$
$$\nu_t(a) = \max_{b \in t, \nu(b) < \nu(a)} (\nu_t(b)) + 1 \quad \text{otherwise.}$$
$\nu_t$ simply numbers the attributes in $t$ in increasing order of $\nu$-numbering, starting with 0. It is easy to see that this is a bijection $t \rightarrow |t|$ and that $\nu = \nu_A$.

5.1.4. Definition
For $x \in S$, the set of $t$-states of $x$ is defined by
$$\text{states}_t(x) = \text{range}_t(\nu_t^{-1}(1)) \times \cdots \times \text{range}_t(\nu_t^{-1}(|t|)).$$
The elements of $\text{states}_t(x)$ may be called $t$-aggregates if we take a static view of the domain, $t$-state vectors if we take a dynamic view of the domain, or $t$-tuples if we view the model as a relational database. A tuple $\sigma$ consists of a number of components which are addressed as $(t)_i$.
□
We use $\sigma_0, \sigma_1, \ldots,$ as metavariables over states. Context will make clear with respect to which type the vector is taken.

The components of the tuples in $\text{states}_t(x)$ are placed in the unique order imposed by $\nu$.

We have defined $\text{states}_t(x)$ for arbitrary $t$ and $x$. In general, we are interested only in the state
vectors containing values other than NA, i.e. state vectors containing values of applicable attributes. All these values are contained in the longest state vector of an object, defined as follows.

5.1.5. Definition
The state space of $x$ is the set

$$\text{states}(x) = \text{states}_{\text{type}}(\{x\})(x).$$

The elements of $\text{states}(x)$ are called the longest state vectors of $x$. $\square$

The term longest state vector is not completely accurate, for the longest state vectors are of course the elements of $\text{states}_A(x)$. The term should be read as short for "longest significant state vector."

We can now define the concept of an object.

5.1.6. Definition
An object is a pair $(x, \sigma)$ with $x \in S$ and $\sigma \in \text{states}(s)$. We use

$$o_0, o_1, \ldots,$$

as metavariables over objects. If $o$ is the object $(x, \sigma), x$ is called the identity of the object, written $\text{id}(o) = x$, and $\sigma$ is called the internal state of the object, written $\text{st}(o) = \sigma$.

A class is a set of objects. $\square$

An object may have a very complex structure. Any of the components of its state vector may be a set of surrogates and any component may be the identity of another object. Moreover, this may be realized in the same component, i.e. if $o_0 = (x_0, (\ldots, x_1, \ldots))$, we may have that $x_1 = \{y_0, \ldots, y_n\}$ and there may be another object $o_1 = (x_1, (z_0, \ldots, z_n))$, and so on recursively. Note that the wff

$$(x_1, (z_0, \ldots, z_n)) = (\{y_0, \ldots, y_n\}, (z_0, \ldots, z_n))$$

is a strict identity.

It is interesting to look at real-world objects displaying this complex structure. Some examples which come to mind are:

1. A gas is a finite set $\{s_0, \ldots, s_n\}$ of molecules which has certain properties $(s_{n+1}, \ldots, s_{m+n})$. Some of its properties are volume, pressure and temperature. These would be deducible from the properties of the individual gas molecules $s_i$, $i = 0, \ldots, n$ via statistical gas laws, but no data are kept about these molecules and they are not named individually. They are therefore not represented by surrogates.

2. A farmer may posses a set $\{s_0, \ldots, s_n\}$ of animals which are named individually. The set itself may have certain properties $(s_{n+1}, \ldots, s_{m+n})$ such as feeding cost, total investment and number of visits of the animal doctor which are not deducible from the properties of the elements of the set. (There may be discounts on large amounts of food and on one visit the doctor may treat several animals.) Note that if an animal is added or removed from the set, a new set which is the identity of a different object comes into being. The bearer of these changes could be defined as the identity of the farm object, which has an attribute animals.
A democratically organized project may have an attribute *members* whose value is the set of employees working on the project. This set has certain properties, such as average salary or the number of people currently working on the project. These properties are deducible from the properties of the elements (average salary) or are defined analytically for the set itself (cardinality).

These examples illustrate some of the design considerations for objects. First, there is the difference between sets and aggregates. In general, the elements of a set do not play distinguishable roles in the set, whereas the components of an aggregate do. The role of a component in an aggregate is indicated intuitively by its name and represented formally by the set of functions which is the attribute.

Second, a sign that a set has its elements outside $S$ is that we do not have names for its elements. The gas in the example is a case in point.

Third, there may be a necessary relation between the properties of a complex surrogate and those of its elements. Examples are the average salary of employees in a set and the total amount of food (as opposed to the feeding cost) for a set of animals. We are not used to viewing these properties as properties of a set, because we usually compute them from the properties of the elements. But of course, they really are properties of the set of elements and not an individual element.

To keep track of the diversity of objects, we introduce names for the different ways in which an object may be structured.

5.1.7. Definition

1. An identity $s$ is called *primitive* if $s \cap S = \emptyset$. $s$ is called a *set identity* if $s \cap S \neq \emptyset$.
2. $s$ is called *atomic* if $\text{type}(s) = \emptyset$. $s$ is called *aggregate* if $\text{type}(s) \neq \emptyset$.
3. $s$ is called *compound* if $s \subseteq S \times \ldots \times S$.

An object is called primitive, set, aggregate, atomic, or compound if its identity is primitive, set, aggregate, atomic, or compound. A attribute is called primitive, set, aggregate, atomic, or compound if its values are primitive, set, aggregate, atomic, or compound.

5.1.8. Definition

The *class* of type $t$ is the set of objects

$$\text{class}(t) = \{(x, \sigma) \mid x \in \text{kind}(t) \land \sigma \in \text{states}(x)\}.$$

If the class is non-empty, $\text{kind}(t)$ is a non-empty natural kind. The set $\text{kind}(t)$ contains precisely the identities of the objects in $\text{class}(t)$.

5.1.9. Definition

The objects of a kind are

$$\text{obj}(k) = \{(x, \sigma) \mid x \in k, \sigma \in \text{states}(x)\}.$$

Obviously, the objects of a natural kind are the set
5.1.10. Definition

The universe $U$ is defined as the objects of kind $S$,

$$U = \{(x, \sigma) \mid x \in S \land \sigma \in \text{states}(x)\}.$$  

The universe thus contains for each surrogate $x \in S$ all objects whose identity is $x$ and have a longest state vector meaningful for $x$. Note that in a strict sense, this is not the set of all possible objects. If $(x, \sigma_0) \in U$, we may form an object $(x, \sigma_1)$ such that $\sigma_1$ contains some, but not all components of $\sigma_0$. Then $\sigma_1$ is not a longest state vector of $x$ and $(x, \sigma_1) \notin U$.

5.1.11. Definition

Let $o_1, o_2 \in U$. Then $o_1$ and $o_2$ are identical when $id(o_1) = id(o_2)$. $o_1$ and $o_2$ are distinguishable if $st(o_1) \neq st(o_2)$.

Two objects are distinguishable if they are of different largest types, or if they are of the same largest type but have different values in their state vectors. The principle of Leibniz cited in section 3.1.3 says that two objects are identical if they are indistinguishable. We there said that this principle cannot hold in a domain when the number of attributes is insufficient to distinguish different objects in all possible cases. We can now explicate the principle and its refutation precisely in terms of objects. Leibniz' principle says that

$$\forall o_1, o_2 \in U[st(o_1) = st(o_2) \Rightarrow id(o_1) = id(o_2)].$$

Obviously, this implication is wholly contingent in our model, for we can have that

$$id(o_1) \neq id(o_2) \text{ but } st(o_1) = st(o_2).$$

The converse of the principle of Leibniz says that identicals are indistinguishable,

$$\forall o_1, o_2 \in U[id(o_1) = id(o_2) \Rightarrow st(o_1) = st(o_2)].$$

This principle cannot be adopted either when we work with possible worlds. In different possible worlds, the same object may occur in different states so that we can have that

$$id(o_1) = id(o_2) \text{ but } st(o_1) \neq st(o_2).$$

In other words, we can talk about two different objects, $o_1 \neq o_2$, which are identical, $id(o_1) = id(o_2)$.

5.2. State spaces and projection functions
5.2.1. Definition
The $t$-state space of $k \subseteq S$ is
\[
ST_t(k) = \bigcup_{x \in k} states_t(x).
\]

Just as for individual identities, we want to restrict the $t$-state space of $k$ to attributes applicable to all elements of $k$. That is, we take $\text{type}(k)$ for $t$. This may be the empty set of attributes, though in general we consider natural kinds for which $\text{type}(k) \neq \emptyset$.

5.2.2. Definition
The state space of $k \subseteq S$ is the set
\[
ST(k) = ST_{\text{type}(k)}(k).
\]

We need one more concept of state space, the state space $ST$ of $S$. This is not the same as $ST(S)$ but consists of all longest state vectors of all identities. In contrast, the set $ST(S)$ contains the state vectors containing values for attributes applicable to all identities. In most domains, $ST(S)$ will be empty.

5.2.3. Definition
The state space of a domain is the set
\[
ST = \bigcup_{x \in S} states_{\text{type}(x)}(x).
\]

To show that all definitions are well-formed, we define projection functions between state spaces and prove that projections are commutative.

5.2.4. Definition
The projection $\pi_{t_1,t_2} : states_{t_2}(s) \to states_{t_1}(s)$ is defined by
\[
(\pi_{t_1,t_2}(t))_i = t_{v_{t_2},v_{t_1}^{-1}}(i) \quad \text{for} \quad i \in states_{t_1}(s) \quad \text{and} \quad i = 1, \ldots, |t_2|.
\]

For example, let $t_1 = \{\text{name}\} \subseteq \{\text{emp#}, \text{name}\} = t_2$, and
- $v_{t_1} : \text{name} \to 1$,
- $v_{t_1} : \text{emp#} \to 1$ and
- $v_{t_1} : \text{name} \to 2$.

If $u = (2, \text{Smith})$, then
\[
(\pi_{t_1,t_2}(u))_1 = (u)_{v_{t_2},v_{t_1}^{-1}}(1)
= (u)_{v_{t_2}}(\text{name})
= (u)_2
= \text{Smith}.
\]

Thus, $\pi_{t_1,t_2}(u) = (\text{Smith})$, so that $\pi_{t_1,t_2}$ does what we expect it to do.

Figure 7 illustrates the relevant mappings in the general case that $t_1 \subseteq t_2$. The desired
property of the projection is that the bottom square of figure 7 is commutative.

5.2.5. Theorem
Let \( x \in S \) and \( t_1 \sqsubseteq t_2 \). Then
\[
\pi_{t_2, t_1}[\text{states}_{t_2}(x)] = \text{states}_{t_1}(x).
\]

Proof.
Let \( n_1 = |k_1| \) and \( n_2 = |k_2| \) and \( x \in S \). If \( t = (x'_1, ..., x'_{n_2}) \in \text{states}_{k_1}(x) \), then
\[
\pi_{t_2, t_1}(t) = (x'_{v_1, v_1^{-1}(1)}, ..., x'_{v_1, v_1^{-1}(1)}) \in \text{states}_{k_1}(x).
\]
Conversely, if \( (x_1, ..., x_{n_1}) \in \text{states}_{k_1}(x) \), then there is a \( t \in \text{states}_{k_2}(x) \) with
\( \pi \rho_{x_j} (t) = (x_1, \ldots, x_n) \), for define \( t = (x'_1, \ldots, x'_m) \) with
\[
x'_j = x_{\rho_{i_1}^{-1}(j)} \text{ if } \rho_{i_1}^{-1}(j) \leq n_1,
\]
\[
x'_j \text{ arbitrary in } range_{\rho_{i_1}^{-1}(j)}(x) \text{ otherwise.} \]

We can thus view a tuple as a tuple of different types and for each type project it to the suitably shortened tuple belonging to that type. The theorem can be extended easily to state spaces.

5.2.6. Theorem
For \( k \subseteq S \) and \( t_1 \subseteq t_2 \),
\[
\pi_{t_1, t_2} [ST_{t_2}(k)] = ST_{t_1}(k).
\]

Proof.
\[
\pi_{t_1, t_2} [ST_{t_2}(k)] = \pi_{t_1, t_2} [\bigcup_{x \in k} states_{t_2}(x)] = \bigcup_{x \in k} \pi_{t_1, t_2} [states_{t_2}(x)] = \bigcup_{x \in k} states_{t_1}(x) = ST_{t_1}(k).
\]

5.3. Worlds

5.3.1. Definition
Let \( w : S \to ST \). For each \( a \), we define the function \( a_w : dom(a) \to S \) by
\[
a_w(x) = \pi_{type(x)}(a)(w(x)), x \in dom(a).
\]
\( \pi_{type(x)}(a) \) simply projects the tuples in \( states(x) \) onto the component corresponding to attribute \( a \).

A function \( w : S \to ST \) assigns one state to each identity and is actually a set
\[
\{(s_0, \sigma_0), (s_1, \sigma_1), \ldots\}
\]
with the following to properties:
1. All objects in the set have different identities;
2. Each identity has exactly one state.

After the addition of a construct which separates existing from nonexistent objects, a function \( w \) will be called a world.

By the very choice of \( S \) as the set of actual and possible objects, we have committed ourselves to the equivalent of an existence predicate in modal logic, an existence set.\(^{13}\) To exist in \( w \) is to be element of the existence set of \( w \).

There are three ways to introduce existence sets in the current set-theoretical framework. The first is to let \( w \) be a partial function and declare \( dom(w) \) to be the (identities of) existing objects. The second is to introduce an entity \( NE \) for "non-existent" and define \( w \) as a total function.

\(^{13}\) The reasons for introducing an existence predicate in modal logic have been summarized neatly in Gamut [1982]. See also Hughes and Cresswell [1968].
and declare $w^{-1}(NE)$ to be the set on non-existing objects (we drop the tedious "identities of objects" from now on). The third solution is to introduce an object $(em, a)$ called an existence monitor, whose state vector has only one component, which in each world is the set of existing objects in that world.

In the first solution, $w$ and $a_w$ are all partial functions. In the second, $w$ is total but $a_w$ partial, and in the third, all functions are total. The first and third solutions are more coherent than the second, and the third also is in harmony to let the functions in each attribute (which now emerge as the functions $a_w$) be total. We therefore opt for the third solution.

Using a selected surrogate to monitor the existing surrogates will also turn out to be convenient when we define insertion and deletion of objects. These are state transitions in the life of the existence monitor, and not (or at least only marginally) in the life of the object inserted or deleted. Note that in the third solution, $a_w(x)$ has no meaning if $x$ is not in the existence set and we must quantify, mostly, over the existence set instead of over $S$.

A minor problem with the use of existence sets which emerges in all three solutions is that it cannot be described whether or not instantiated objects (objects whose identity is added to the existence set) have existed before. Therefore, we cannot express integrity constraints which forbid reincarnation. This is a minor problem, for it can easily be solved by working with two sets, containing present and past objects. This does not provide us any new insights, so we do not follow this route here. Moreover, in a future report we will introduce histories by defining objects as pairs $(s, h)$, where $h$ is a sequence of state vectors containing the history of $s$. Deletion will then be represented by the addition of a final tuple to the history. The existence set will then contain all present and past objects, and present objects are then defined as objects whose identities are in the existence set and whose histories are not yet finished. The necessary distinctions can thus be made using a single existence set and here we anticipate upon this future extension.

A major problem with all three solutions is that a basic principle of set theory is that, "given a collection $M$ of mathematical objects, subcollections are themselves perfectly reasonable mathematical objects, as are collections of these new objects, and so on." (Barwise [1975], p. 7) If to be a reasonable mathematical object is to exist, we are in trouble, for the existence set (set?) of a world then becomes very large. And in the current Platonic view, it is perfectly reasonable to say that abstract objects exist, independently of human observation, if they are reasonable mathematical objects. (Bernays [1935] regards the view that mathematical objects exist independently of the reflecting subject as the hallmark of Platonism.) Each existence set would then in effect be a model of axiomatic set theory. Obviously, we must cut this route short, or the specification of database updates will become unmanageably infinite. For example, if we delete an object $o$ from the existence set, we would have to delete all objects in which $o$ occurs, and in the set-theoretical universe there are in general nonenumerably infinitely many of these.

We therefore view the existence set not so much as the collection of objects which exists, but which is actual as opposed to possible. And to be actual will be defined below as to be the value of an attribute (or to be an element of an actual object). In other words, if in $w$ a project
has members \( \{e_0, e_1, e_2\} \), then only that set is said to exist, but if \( s_0 = \{e_0, e_1, e_2\} \) is not a value of any attribute, then \( s_0 \) is said not to exist. On the other hand, because \( \{e_0, e_1, e_2\} \) exists, \( e_0, e_1 \) and \( e_2 \) are required to exist as well. We now formalize these ideas.

**5.3.2.**

We add the following axioms for the constants \( ext \) and \( em \) to \( \Sigma_S \) and \( \Sigma_A \):

\[
em \in S, \\
\ext = \{\{em\} \rightarrow \mathcal{F}(S)\}.
\]

\( \Sigma_A \) also contains uniqueness axioms for \( ext \) (see appendix 3). The reason for demanding that 
\( em \in S \) will become clear in a moment.

To be able to talk about \( ext_w(em) \) as the value of \( ext \) in \( w \), we consider functions 
\( w: S \cup \{em\} \rightarrow ST \). For each such function we can then talk of the identities which exist in \( w \), 
\( ext_w(em) \). Each of these identities has a state \( w(x) \) in \( w \). Non-existing identities also have a state, but this is not relevant for these are never used.

**5.3.3. Definition**

Let \( w \subseteq U \) be a function \( S \cup \{em\} \rightarrow ST \) with \( w(x) \in states(x) \). \( w \) is called a world if it satisfies the following existence requirements:

1. \( a_w [ext_w(em) \cap \text{dom}(a)] \subseteq ext_w(em) \) and
2. \( \forall x \in ext_w(em) [x \cap S = x \cap ext_w(em)] \).
3. \( ext_w(em) \cap S^n \subseteq (ext_w(em))^n \).

The set of all functions satisfying these requirements is called the set \( PW \) of possible worlds.

We use 
\( w_0, w_1, ..., \)

as metavariables over the set of possible words. \( \square \)

The first requirement says that attribute values of existing aggregate objects exist. If these values have themselves attributes, their values are required to exist as well, etc.

The second requirement demands that the elements of existing set identities exist as well. If those elements are set identities, then their elements are required to exist as well, etc.

The third requirement demands that the components of existing tuple identities exist as well. Like the other two, this demand is recursive.

These requirements are the translation of referential integrity constraints from relational theory into a set-theoretical framework.

If we would have demanded that \( em \in S \), then requirement 1 would have implied that 
\( ext_w[ext_w \cap \text{dom}(ext)] \subseteq ext_w(em) \). If \( em \in ext_w(em) \), this is true, but if \( em \in ext_w(em) \) (which is allowed), it reduces to

\( ext_w[\{em\}] \subseteq ext_w(em) \),

which implies \( ext_w(em) \in ext_w(em) \). This violates regularity. There are two ways to avoid this, either by demanding that \( em \in ext_w(em) \) for all \( w \), or by demanding that \( em \in S \). We
have chosen the second way, though nothing much hinges on the choice between one rather than
the other of these two options.

5.3.4.

Note that we do not demand that $\text{ext}_w(\text{em})$ be the smallest set satisfying the existence require­
ments. To see why we do not do this, recall that atomic objects have a zero-length state vector,
but may contain elements, i.e. be sets (see definition 5.1.7). In our example, the set $\text{members}_w(p)$ is atomic, for no attribute applies to it.

If $\text{ext}_w(\text{em})$ were the smallest set satisfying the existence requirements, then we would
have to explicitly describe the addition and deletion of atomic objects according to whether
they become or cease to be attribute values or elements of existing objects. But we do not like
to describe the addition and deletion of a natural number every time it becomes and ceases to
be an attribute value or element of an existing object. Moreover, in a Platonic view atomic
objects should always be in the existence set, for unchangable objects exist eternally.

Socrates - “So then, are not the compounded and the composite naturally liable to be
composed in the same way in which they were put together? And if there is anything
uncompounded is not that, if anything, naturally immune to decomposition?”

We could therefore add the requirement that all atomic objects be in the existence set,

4. $\text{type}(x) = \emptyset \Rightarrow x \in \text{ext}_w(\text{em}).$

But now we have invited the vast set-theoretical universe back into the existence set, for all sets
to which no attributes apply are atomic. This may not be a problem, for no updates have to be
specified for these sets (they have no attributes to be updated), but it does not sit comfortably
next to the frugality with which we decided to manage the existence of non-atomic objects. So
let us restrict 4 to our version of Urelemente:

4'. $\text{type}(x) = \emptyset \land x \cap S = \emptyset \Rightarrow x \in \text{ext}_w(\text{em}).$

But now we have excluded too much from the existence set, for if $\mathbb{N} \subset S,
\forall n \in \mathbb{N} \left( (n+1) \cap S = n \right)$. We end up with only $\emptyset$ being the identity of an eternally existent
object. As a last resort, we can simply demand ad hoc that any sort we please be in $\text{ext}_w(\text{em}),
e.g. \mathbb{N} \subset \text{ext}_w(\text{em})$ if the domain contains natural numbers. If we choose this kludge, we must
tailor the definition of worlds to the particular domain theory we are working with by adding
requirements $\text{sn} \subset \text{ext}_w(\text{em})$ for sort names $\text{sn}$, and can then demand that $\text{ext}_w(\text{em})$ be the
smallest set satisfying the existence requirements. Although this alternative seems workable, we
prefer to leave the options open as long as we can, which is why we omitted any reference to it
in the definition of worlds.

---

14. Plato, *Phaedo* par. 788B. This is similar to the view expressed by Leibniz in his *Monadology* cited ear­
ier.

15. From a Buddhistic standpoint, this is the correct view to take.
5.4. Extensions

5.4.1. Definition

The extension of a kind \( k \) in world \( w \) is the set
\[
    k_w = k \cap \text{ext}_w(\text{em}).
\]
The extension of a type \( t \) in \( w \) is the extension of its kind in \( w \),
\[
    \text{ext}_w(t) = \text{kind}(t) \cap \text{ext}_w(\text{em}).
\]

The definition clearly distinguishes the kind of a type from its extension. "Kind" is an intensional concept, whereas extension is an extensional concept. Each type determines the same kind in all possible worlds, but its extension depends upon the state of the world.

Note that we use the same symbol, \( \text{ext}_w \), to denote the attribute of \( \text{em} \) and the extension of a type in \( w \).

Not all properties of kinds are inherited by extensions. We are interested, first, in the properties that are inherited and second, in the properties of an extension which can tell us something about the kinds of which it is an extension. The following list of theorems compares each property of kinds with the corresponding property of extensions. The list is followed by a summary of the answers to these two questions.

We use the following two basic facts about extensions.

\[
k_w \subseteq k
\]
\[
type(k) \subseteq type(k_w).
\]
The first follows from the definition of \( k_w \) and the second follows from the first.

The left column of each theorem gives the theorem for kinds corresponding to the theorems for extensions in the right column. The numbers of the theorems for kinds are added.

5.4.2. Theorem

<table>
<thead>
<tr>
<th>1. (4.1.4.1)</th>
<th>( \text{kind}(\emptyset) = S )</th>
<th>( \text{ext}_w(\emptyset) = \text{ext}_w(\text{em}) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. (4.1.4.3)</td>
<td>( \text{kind}(t) = \bigcap_{a \in t} \text{dom}(a) )</td>
<td>( \text{ext}<em>w(t) = \bigcap</em>{a \in t} \text{ext}_w({a}) )</td>
</tr>
<tr>
<td></td>
<td>( \text{ext}<em>w(t) \subseteq \bigcap</em>{a \in t} \text{dom}(a) )</td>
<td></td>
</tr>
<tr>
<td>3. (4.1.4.4)</td>
<td>( k \subseteq \text{kind}(t) \Rightarrow t \subseteq \text{type}(k) )</td>
<td>( k_w \subseteq \text{ext}_w(t) \Rightarrow t \subseteq \text{type}(k_w) )</td>
</tr>
<tr>
<td>4. (4.1.4.5)</td>
<td>( t_1 \subseteq t_2 \Rightarrow \text{kind}(t_2) \subseteq \text{kind}(t_1) )</td>
<td>( t_1 \subseteq t_2 \Rightarrow \text{ext}_w(t_2) \subseteq \text{ext}_w(t_1) )</td>
</tr>
<tr>
<td>5. (4.1.4.6)</td>
<td>( k_1 \subseteq k_2 \Rightarrow \text{type}(k_2) \subseteq \text{type}(k_1) )</td>
<td>( k_1 \subseteq k_2 \Rightarrow \text{type}(k_2) \subseteq \text{type}(k_1) )</td>
</tr>
</tbody>
</table>

Proof

1 and 2 are trivial. For 3, note that by definition, \( \text{ext}_w(t) \subseteq \text{kind}(t) \), so
\[
t \subseteq \text{type}(\text{kind}(t)) \subseteq \text{type}(\text{ext}_w(t)).
\]

Then
\( k_w \subseteq \text{ext}_w(t) \Rightarrow \text{type}(\text{ext}_w(t)) \subseteq \text{type}(k_w) \Rightarrow t \subseteq \text{type}(k_w). \)

For the other half, we use 4.1.5.2 to note that
\[
k_w \subseteq \text{kind}(\text{type}(k_w)), \quad \text{so}
\]
\[
k_w \cap \text{ext}_w(em) \subseteq \text{kind}(\text{type}(k_w)) \cap \text{ext}_w(em), \quad \text{so}
\]
\[
k_w \subseteq \text{ext}_w(\text{type}(k_w)).
\]

Therefore,
\[
t \subseteq \text{type}(k_w) \Rightarrow \text{ext}_w(\text{type}(k_w)) \subseteq \text{ext}_w(t) \Rightarrow k_w \subseteq \text{ext}_w(t).
\]

In this proof, we used 4 and 5, which are proved independently.

4. \( t \subseteq t_2 \Rightarrow \text{kind}(t_2) \cap \text{ext}_w(em) \subseteq \text{kind}(t_1) \cap \text{ext}_w(em) \).

5. 4.1.4.6 is a general statement for all kinds. ∎

Thus, all parts of theorem 4.1.4 can be relativized to worlds. This means that generalization relations are invariant in all possible worlds. For example, 3 says that if all existing truck drivers in \( w \) are employees in \( w \), then all employee attributes are applicable to truck drivers, and vice versa. 4 says that specialization indices the same subset hierarchy in every possible world.

### 5.4.3. Theorem

| 4.1.5.1 | \( t \subseteq \text{type}(\text{kind}(t)) \) | \( t \subseteq \text{type}(\text{ext}_w(t)) \) |
| 4.1.5.2 | \( k \subseteq \text{kind}(\text{type}(k)) \) | \( k_w \subseteq \text{ext}_w(\text{type}(k_w)) \) |

**Proof.**

This has been proven in 3 above. ∎

The definition of natural kinds has a pleasing consequence for extensions of a natural kind, as is shown in the next theorem.

### 5.4.4. Theorem

| 1. \( k = \text{kind}(\text{type}(k)) \) | \( k_w = \text{ext}_w(\text{type}(k_w)) \) |
| 2. \( t = \text{type}(\text{kind}(t)) \) | - |

**Proof.**

\[
k_w \subseteq k \Rightarrow 
\]
\[
\text{kind}(\text{type}(k_w)) \subseteq \text{kind}(\text{type}(k)) \Rightarrow 
\]
\[
\text{kind}(\text{type}(k_w)) \subseteq k \Rightarrow 
\]
\[
\text{kind}(\text{type}(k_w)) \cap \text{ext}_w(em) \subseteq k \cap \text{ext}_w(em) \Rightarrow 
\]
\[
\text{ext}_w(\text{type}(k_w)) \subseteq k_w. \quad \square
\]

To give a counterexample of \( t = \text{type}(\text{ext}_w(t)) \), let \( r_0 = \text{type}(Emp) \), then in \( w_0 \) the extension of \( r_0 \) may happen to consist only of truck drivers. Then the type of this extension is larger than
As an illustration of 1, if \( \mathcal{E}_0 \) is the extension of the natural kind of employees in world \( w_0 \), then \( \text{type}(\mathcal{E}_0) \) is a superset of \( \text{type}(\text{Emp}) \). The cases where the existing employees are all of a genuine subkind of \( \text{Emp} \) -for example, if they all employees in \( \mathcal{E}_0 \) truck drivers- are precisely the cases where \( \text{type}(\text{Emp}) \subseteq \text{type}(\mathcal{E}_0) \). But even if we don’t know what smallest natural kind \( \mathcal{E}_0 \) is an extension of, we do know that the current extension of that natural kind is \( \mathcal{E}_0 \).

This equality fails for unnatural kinds. For example, take a domain where employees are either secretary or engineer or truck driver and let \( k_1 \) be the set currently existing secretaries and engineers. The type of \( k_1 \) is the type of employees, and the extension of that type includes the currently existing secretaries and engineers as well as the currently existing truck drivers, so that

\[ k_1 \neq \text{ext}_w(\text{type}(k_1)). \]

Theorem 4.1.7 says that a type is a natural type iff it is the type of a set of identities, so that an easy test whether a type is natural is to see if it is the set of all attributes applicable to a set of existing identities \( k_w \) in an arbitrary world \( w \). Though we are guaranteed to find only natural types in this way, the counterexample above shows that we will in general not find all natural types in this way.

5.4.5. Theorem

<table>
<thead>
<tr>
<th>1. (4.1.8.3)</th>
<th>( \text{kind}(\ell) \subseteq \mathcal{E} \Rightarrow \text{type}(\mathcal{E}) \subseteq \ell )</th>
<th>( \text{type}(\mathcal{E}) \subseteq \ell \Rightarrow \text{ext}_w(\ell) \subseteq \mathcal{E}_w )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. (4.1.8.4)</td>
<td>( \mathcal{E}_1 \subseteq \mathcal{E}_2 \Rightarrow \text{type}(\mathcal{E}_2) \subseteq \text{type}(\mathcal{E}_1) )</td>
<td>( \mathcal{E}_1 \subseteq \mathcal{E}_2 \Rightarrow \text{type}(\mathcal{E}_2) \subseteq \text{type}(\mathcal{E}_1) )</td>
</tr>
<tr>
<td>3. (4.1.8.5)</td>
<td>( \ell_1 \subseteq \ell_2 \Rightarrow \text{kind}(\ell_2) \subseteq \text{kind}(\ell_1) )</td>
<td>-</td>
</tr>
</tbody>
</table>

Proof.
1. \( \text{type}(\mathcal{E}_w) \subseteq \ell \Rightarrow \text{ext}_w(\ell) \subseteq \text{ext}_w(\text{type}(\mathcal{E}_w)) = \mathcal{E}_w \).
2. \( \text{type}(\mathcal{E}_1) \subseteq \text{type}(\mathcal{E}_2) \Rightarrow \text{ext}_w(\text{type}(\mathcal{E}_1)) \subseteq \text{ext}_w(\text{type}(\mathcal{E}_2)) = \mathcal{E}_1 \subseteq \mathcal{E}_2 \).

To show that \( \text{ext}_w(\ell_2) \subseteq \text{ext}_w(\ell_1) \neq \mathcal{E}_1 \subseteq \ell_2 \),

take the case that

\[ \text{ext}_w(\{\text{emp#}\}) \subseteq \text{ext}_w(\{\text{emp#}, \text{licence#}\}), \text{ but } \{\text{emp#}, \text{licence#}\} \notin \{\text{emp#}\}. \]
5.4.6. Theorem

<table>
<thead>
<tr>
<th></th>
<th>(4.1.10.1)</th>
<th></th>
<th>(4.1.10.2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>[ k \text{ is a natural kind iff } type(k) = type(k') \models k' \subseteq k ]</td>
<td>If ( k ) is a natural kind then [ type(k_w) = type(k'_w) \models k'_w \subseteq k_w ]</td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>[ t \text{ is a natural type iff } \text{kind}(t) = \text{kind}(t') \models t' \subseteq t ]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Proof.

If \( k \) is a natural kind, then

\[
\begin{align*}
type(k_w) &= type(k'_w) \\
ext_w(type(k_w)) &= ext_w(type(k'_w)) \\
k_w &= ext_w(type(k'_w)) \\
k'_w &\subseteq k_w.
\end{align*}
\]

A counterexample to the converse of 1 is a world \( w \) where

- \( k_w \) is the set of actual secretaries and engineers, and
- all actual employees are secretaries or engineers.

Then \( type(k_w) = type(k'_w) \) only if \( k'_w \) is a set a some actual secretaries or engineers, so

\[
type(k_w) = type(k'_w).
\]

But \( k_w \) is not the extension of a natural kind.

Nothing can be proved in the case of 2. Take for example a world in which all and only the actual temporary employees are secretaries. These have natural types and equal extensions, but neither is contained in the other. Conversely, two arbitrary attribute sets, one of which is contained in the other, may have empty extensions in a world. Then

\[
\text{kind}(t) = \text{kind}(t') \models t' \subseteq t,
\]

but neither type is natural.

5.4.7. Theorem

<table>
<thead>
<tr>
<th></th>
<th>(4.2.2.1)</th>
<th></th>
<th>(4.2.2.2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>[ \bigcap_i \text{kind}(t_i) = \text{kind}(\bigcup_i t_i) ]</td>
<td>[ \bigcap_i ext_w(t_i) = ext_w(\bigcup_i t_i) ]</td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>[ \bigcup_i \text{kind}(t_i) \subseteq \text{kind}(\bigcap_i t_i) ]</td>
<td>[ \bigcup_i ext_w(t_i) \subseteq ext_w(\bigcap_i t_i) ]</td>
<td></td>
</tr>
</tbody>
</table>

Proof.

Immediate. \( \square \)
5.4.8.

Instead of looking at the identities of existing objects, as we have done up till now, we can also study their states. For example, the states of existing objects of kind \( k \) are the set \( w[k_w] \). This set contains for each identity in \( k_w \) its longest state vector. Viewed as a set of objects, \( w[k_w] \subseteq w \). For a natural kind, \( w[k_w] \) is the analogon in the current object-oriented framework of the relation instance of relational theory.

Let \( w^*(x) \) be \( w(x) \) padded with \( NA \)'s to give it the length \( |A| \). Then \( w^*[ext_w(em)] \) is the set of \( A \)-vectors of all existing objects. Viewed as a set of objects, it corresponds to the universal relation instance of relational theory. A comparison with relational theory is done in section 7.1.

The projection theorem 5.2.6 can be relativized to the state sets of existing objects.

5.4.9. Theorem

If \( t_1 \sqsubseteq t_2 \), \( \pi_{t_1} [\pi_{A, I_2} [w^*[k_w]]] = \pi_{A, I_1} [w^*[k_w]] \)

Proof.
Omitted. □

The set \( \pi_{A, I_2} [w^*[k_w]] \) is the set of \( t_2 \)-vectors of identities in \( k_w \); the set \( \pi_{A, I_1} [w^*[k_w]] \) is the set of \( t_1 \)-vectors of the same identities. Obviously, \( \pi_{t_1} \) projects the one onto the other.

5.4.10.

We now summarize the answers to the two questions

1. Which properties of kinds are inherited by their extensions and
2. Which properties of natural kinds can we deduce from those of their extensions.

We found that the kind and type lattices are preserved by extensions and that the type- and extension lattices are each other's inverse. Together with the relativization of 4.2.2, this means that in each world the extensions of the types in each specialization group are disjoint and form a partition if the specialization group is exhaustive.

The criteria for natural kinds in terms of their extensions are purely negative, i.e. they are of the form

If \( k \) is a natural kind, then \( \phi(k) \).

Using this, we can only show definitively that something is not a natural kind. We found three negative criteria:

1. \( k_w = type(ext_w(k_w)) \).
2. \( type(k_{2_w}) \subseteq type(k_{1_w}) \Rightarrow k_{1_w} \subseteq k_{2_w} \).
3. \( type(k_w) = type(k'_w) \Rightarrow k'_w \subseteq k_w \).

The only positive criterion is that if \( t \) is the type of any set of existing identities, then \( kind(t) \) is a natural kind. There is no criterion to find out whether a type is natural, i.e. whether we have found all attributes applicable to a natural kind.
Chapter 6
Static integrity constraints

6.1. Admissible worlds and the language \( L_5(CON, i) \)

6.1.1.

The set \( PW \) can now be used as a supplier of models for static integrity constraints expressed in \( L_5(CON) \). A static constraint like "A project must have at least one member" can be expressed in \( L_5(CON) \) as \( x \in \text{Project}_w[\text{members}_w(x) \neq \emptyset] \). This is an open wff which is satisfied by some \( w \in PW \). Those worlds which satisfy the static integrity constraints are called admissible worlds. Definitions follow below.

To get a feeling of the status of \( PW \) and the integrity constraints, consider a classification of physical laws given by van Fraassen [1970].

"In the case of a nonrelativistic theory, the function of a law is to describe the behavior of the kind of physical system with which the theory deals: to describe the possible states of which it is capable, its normal evolution through time when undisturbed, and its behavior in interaction. We shall therefore proceed in accordance with the traditional threefold distinction between laws of coexistence, laws of succession, and laws of interaction. (p. 330)"

Of each of these, there are statistical and nonstatistical versions. Van Fraassen continues: "Laws of coexistence select the physically possible subset of the state-space... Laws of succession select the physically possible trajectories in the state-space." (p. 330-331) Laws of interaction describe how a system behaves when it is aggregated with other systems into a larger system. Examples are the familiar Boyle-Charles gas law \( PV = RT \) (law of coexistence), the classical \( p = m.v \) (law of succession) and the composition of laws describing individual particles in many-body systems (law of interaction).

Obviously, van Fraassen's laws of coexistence are the static integrity constraints of database domain modelling and laws of succession and interaction are dynamic constraints. We postpone dynamic constraints to a forthcoming report and concentrate now on static constraints.

6.1.2. Definition

A wff \( \phi(w) \) containing \( w \) as only free variable is called a static integrity constraint. Let \( IC(w) \) be a conjunction of wff's containing \( w \) as only free variable. Then the set \( AM \) of admissible worlds selected by \( IC \) is

\[
AM = \{w \in PW \mid IC(w)\}.
\]

In order to formulate the static constraints easily, we extend \( L_D \) with the description operator. The resulting language is called \( L_5(CON) \).
6.1.3. Definition

The terms of \( L_S(CON, \iota) \) are defined as follows.

1. If \( t \) is an individual variable or constant, \( t \) is a term.
2. If \( x \) is an individual variable and \( \varphi \) a wff, \( \forall x[\varphi] \) is a term. \( \square \)

The new syntax is given an interpretation as follows.

6.1.4. Definition

\[
I(\forall x[\varphi]) = s_i \text{ if } \exists i \in \mathbb{N} \exists x \in S[\varphi(x)] \land x = s_i, \\
\emptyset \text{ otherwise. } \square
\]

Thus, \( \emptyset \) plays the role of the distinguished element in the domain which indicates absence. Note that \( \forall x[\emptyset] \) is regarded as being absent if there is more than one \( x \) with \( \emptyset \).

The inference rules of \( L_S(CON, \iota) \) are the same as those for \( L_S(CON) \).

6.2. Examples

6.2.1. Static constraints can be classified into three groups. Existence constraints demand that an object exists (or does not exist), uniqueness constraints demand that precisely one object exists, and coexistence constraints demand that certain objects exist in combination. We treat these in order.

6.2.2. Existence constraints

The most basic existence constraints have been built into our concept of a possible world, that the components of existing compound objects exist. Attribute values of existing aggregates, elements of existing sets, and components of existing tuples are required to exist. Thus, components are essential to compound objects in that they must exist in all possible worlds in which the compound exists.

The three existence requirements for possible worlds are examples of a type of existence constraint which requires the existence of components. Let us call these constraints of required existence. There are other required existence constraints. For example, the three basic existence requirements guarantee that existing projects have existing project members and that existing tasks are tasks of existing employees and projects:

\[
\text{members} = [Project \rightarrow \emptyset(Emp)] \\
\text{task} = [Emp \times Project \rightarrow \text{TASKS}].
\]

But we want to impose the additional constraint that only \( \text{employee, project} \) pairs exist for which \( \text{employee} \in \text{members}_\iota(\text{project}) \). This constraint can be formulated thus:

\[
\forall(x, y) \in (Emp \times Project) \cap \text{ext}_\iota(\text{em})[x \in \text{members}_\iota(y)] \tag{IC1}
\]

Note that a required component may be \( \emptyset \). If this is disallowed, that constraint must be added to \( IC \). We may call such a constraint a presence constraint. An example is
A dual to the required existence of components is the dependent existence of a component. 

\( x \in S \) is said to have dependent existence in \( w \) iff 

\[ x \in ext_w(em) \iff \exists y \in ext_w(em) \exists a[x = a_w(y)]. \]

For example, we may declare adresses to have dependent existence. An address then ceases to exist if there is no existing bearer of the address.

Although existence constraints are easy to formulate, we may expect them to play havoc with the correct specification of update procedures. Think for a moment what is involved in the deletion of objects from a world in connection with each of the types of existence constraint. Furtado et al. [1981] give a detailed analysis of a specification of updates which is correct with respect to one simple existence constraint, which requires attribute values to exist if the aggregate exists.

### 6.2.3. Uniqueness constraints

Uniqueness constraints can be expressed simply by defining the attributes to be sets of 1-1 functions. An example is

\[ emp\# = [Emp \rightarrow^{1-1} EMP\#]. \]

Other examples can be found in appendix 3. These uniqueness constraints are thus not part of the definition of \( AM \) but of \( PW \). There is no metaphysics in this, just convenience.

Uniqueness constraints are a generalization of the concept of key from relational theory. Note that employee numbers, licence numbers, etc. are only required to be unique in each possible world; they need not be so across possible worlds. Two employees can swap employee numbers from one possible world to the next and still respect the uniqueness constraint for \( emp\# \).

A more refined example of a uniqueness constraint is the constraint that in one project, each employee has a unique task:

\[ \forall x \in Project_w[\text{task}_w(x) \times \{x\} \rightarrow^{1-1} TASKS] \]  

(\text{IC4})

Still another example is that each project has exactly one leader:

\[ \forall x \in Project_w \exists ! y \in members_w(x)[\text{task}_w(y, x) = \text{leader}]. \]  

(\text{IC5})

### 6.2.4. Coexistence constraints

All other constraints on possible worlds are lumped together under the term "coexistence constraints." An example is the requirement that each project has at least three members:

\[ \forall x \in Project_w[|members_w(x)| > 2]. \]  

(\text{IC6})
As an example of the use of the description operator, assume that IC 5 holds, so that $\forall x [task_w(x, y) = leader]$ has the semantics “the leader of project $x$ in world $w$.” Suppose that the attribute

$$salary = [Emp \to \mathbb{N}]$$

has been defined and that we have a function

$$avg : \mathcal{P}(\mathbb{N}) \to \mathbb{R}$$

which computes the average of a set of natural numbers. (If arithmetic is only defined for natural numbers, use truncation or rounding and/or multiply the numbers with powers of ten so that we can work solely with natural numbers.) Then the constraint that the average salary of project members does not exceed the salary of the project leader is

$$\forall x \in Project_w [avg(salary_w(members_w(x))) \leq salary_w(\forall y [task_w(y, x) = leader])]. \quad (IC7)$$
7.1. Some problems from philosophical logic

7.1.1. The treatment of set theory in the language of modal logic is a minefield through which one has to tread very carefully. $PW$ can serve as a Kripke model for a theory containing S5 axioms expressed in a modal language, i.e. one containing operators for $\Box$ ("it is necessary that ...") and $\Diamond$ ("it is possible that ..."). The interpretation of $\Box \phi$ is that $\phi$ is true in all possible worlds, whereas $\Diamond \phi$ is true iff $\phi$ is true in at least one possible world (see Hughes & Cresswell [1968] for details). When the static constraints are expressed in such a language, the set $AM$ of admissible worlds is then simply a Kripke model which satisfies the theory consisting of the integrity constraints and the S5 axioms.

More in detail, the language $L_5(CON, \iota)$, extended with operators $\Box$ and $\Diamond$, would have to be interpreted by assigning an interpretation to $\in$ which is independent of the current world, because otherwise we would not be working in a single model for ZF anymore. In the extreme case, each possible world would be a model for ZF.

The attribute names would have to be introduced as function symbols, not as constants, and each attribute name $a$ is interpreted in $w$ as $a_w$. Natural kind names are constants which receive the same interpretation in each possible world.

We will not define a modal language for integrity constraints in detail, for there are a number of conceptual problems in S5 logics which are absent from the formulation of constraints in $L_5(CON, \iota)$. We briefly discuss some of these problems below.

7.1.2. A well-known problem in S5 is that

$$x = y \Rightarrow \Box(x = y)$$

is a theorem. Using this theorem (and using attribute names as function names, as indicated above), we would have

$$members(x) = members(y) \Rightarrow \Box(members(x) = members(y)).$$

In words, if two projects have the same set of members, they necessarily have the same set of members. But if two projects have the same set of members, they need not have so in all possible worlds, so this consequence is counterintuitive. In $L_5(CON)$, however, the equivalent of (1) is

$$x = y \Rightarrow \forall w [x = y]$$

and this is obviously valid. If two variables refer to the same set, they do so independently of the world we are in. And from (2) we are not allowed to infer
members_w(x) = members_w(y) \Rightarrow \forall w [members_w(x) = members_w(y)],

for the quantifier now captures one of the free variables in the terms substituted for x and y.

By quantifying explicitly over worlds, the paradoxes involved in (1) are avoided.

Kripke's [1971], [1980] solution to (1) is to treat constants as rigid designators, i.e. as names which refer to the same identity in every possible world. We effectively did so by naming the elements of S by \mathcal{N} (using the function \pi_i, giving a name to the set of possible interpretations of a function, and by interpreting natural kind names rigidly. (See Quine [...] ..... for the reasons for interpreting natural kind names rigidly).

Note also that we allow for arbitrary differences in the states of objects across possible worlds. The problem of trans-world identification is "solved" by simply declaring it to be solved: The capability to talk of a different state of the same object in different possible worlds presupposes our ability to identify two objects as being two manifestations of an identical object.

7.1.3.

A problem which is related to (1) is that descriptions may not be substituted freely for x in Vx[\phi(x)] if \phi contains a modal operator. For example, let lose(x) be the set of losers of game x and winner(x) be the winner of game x. Then V elimination would lead to the inference

\forall y \Diamond (y \in lose(x))
\Rightarrow (winner(x) \in lose(x)).

It is true that the winner could have lost in another possible world, but it is not true that there is a world in which the winner of x is in the set of losers of x. Here, too, the paradox disappears in L_s(CON, \iota). From

\forall y \exists w[y \in lose_w(x)]

we cannot infer

\forall y \exists w[winner_w(y) \in lose_w(x)],

for the free variable w in winner_w(x) is captured by the quantifier.

These problems can be avoided in S5 by sharpening the inference rules of the logic (see for example Kutschera [1976]). Application to integrity constraints for admissible worlds is a complex matter which will have to wait future research.

7.2. Universal relation models

7.2.1.

Before we can compare the domain axiomatization proposed in this report with other approaches to domain models, we must clear up a terminological problem which obscures a fundamental difference between the approach in this report and other approaches.

In speaking of data models, we use the term "model" in a way different from the ways we have used the term in this report.
1. The first meaning in which we have used "model" is idealization or simplification or abstraction. In this sense, the domain is a model of the UoD and the DB a model of the implementation (figure 1). 16

2. The second meaning of "model" in this report is the Tarskian sense of the word which is now standard in predicate logic. In this sense, the domain is a model for an uninterpreted formal language, $L_5$.

3. In "data model," "model" occurs in a third sense, which can perhaps best be described as class of structures which can be found in the database.

It is important to realize that the third meaning differs in two respects from the second meaning. First, it concerns the database, not the domain, and second, it concerns a class of database structures, not a particular database. When we talk of the relational model or the RM/T model, we use "model" in the third sense. To have a name for the datamodel (third sense) developed in this report, we call it ABSURD, for ABSTRACT Surrogate Domain.

Because a database is a representation of an infinite domain in a finite machine, we may expect a number of structures in databases which are a peculiarity of the finiteness of the database and do not occur in the domain itself. This expectation will be confirmed when we compare database models with ABSURD. For example, in the database we will have to choose between storing or computing attribute values whereas in the domain, there is no difference between these two. If we choose to compute a value in the database, the database will have to support virtual tables and this in its turn entails some decision about the possibility (and semantics) of view updates. If on the other hand we choose to store a value, then we must decide in which relation instance(s) it should be stored. If it is stored in more than one instance, then some sort of referential integrity maintenance must be decided upon. Virtual tables, relational decomposition and referential integrity between tables are database concepts, not domain concepts.

7.2.2.

Universal relation (UR) models have been proposed or criticized, among others, by Atzeni & Parker [1982], Biskup & Brüggemann [1983], Maier, Rozenshtein & Warren [1983], Maier, Ullman & Vardi [1984], Ullman [1982]. We first summarize the central idea and then compare it with ABSURD.

In UR models, the user of the database can view the database as a single relation called the universal relation. A tuple in a UR instance may contain many null values, but since the UR is, in general, not actually stored, these nulls are not stored either. Rather, the database allows the use to ask queries as if the UR were stored, thus freeing the user of "logical navigation," i.e. the specification of which attributes belong to which DB relation. Each attribute occurs only once in the UR, so that the user is also freed of remembering which attributes in different DB relations are the same and which are different.

ABSURD shares a number of assumptions with UR models, notably the unique name
assumption, which says that each attribute has a unique name whose intuitive meaning it is to refer to the same role in each possible context.

UR models also know the relationship uniqueness assumption, which says that, given an arbitrary database state \( d \), there is a unique relation instance \([X](d)\) for each attribute set \( X \). In ABSURD, we also make this assumption, but with a difference. For each \( t \subseteq A \) we have defined a single set \( \text{kind}(t) \), but for some types this set is empty. What we are interested in is the extensions of natural kinds in a world, and these are precisely the kinds of types \( t \) for which \( \text{kind}(t) \) is not empty.

Maier et al. [1984] mention a third assumption which they call the one flavor assumption and which says that the "real world significance" of the tuples in \([X](d)\) is independent of the access path used to compute that relation instance. We do not make this assumption in ABSURD, because it concerns not the domain, but the implementation of the DB. In the (Platonic) domain, there is no difference between storage and computation. But as a statement about the implementation of the DB, it is not so much an assumption as a requirement for the implementation.

The following is a list of differences between the ABSURD approach and UR models.

1. **Level of abstraction.** The UR is an abstraction from the database, not from the domain. The reason why UR theory has "not been picked up, explicitly, by builders of real systems (as opposed to prototypes)" (Ullman [1987], p. 5) may be that it is not natural for the user to view the data in the DB as an instance of one huge relation. Instead, they may find it more natural to look at an abstraction of the UoD, the domain, through the spectacles which the DB provides. As Atzeni & Parker [1982] (p. 1) remark,

   In making assumptions (such as the universal relation assumption) more for establishing certain results (such as the usefulness of 'decomposition' as a design methodology) than for their appropriateness in modelling data, one must realize that the results may not be useful to database engineers.

   The user as well as the database engineer want to view the domain in terms of structures natural to the domain, instead of the database in terms of structures which give rise to neat theorems.

2. **Relation between domain and database.** Much of UR theory has to do with the semantics of the stored DB relation instances in terms of a hypothetical UR instance. This corresponds to the relation between the DB and the domain in ABSURD. UR research has, in my opinion, picked up the stick at the wrong end; instead of starting with the investigation of the domain structures which a DB should represent, it started with the properties of a representation without being clear about what should be represented.

3. **Null values.** One UR instance corresponds roughly to the set \( w^{\text{ext}_w}(\text{em}) \), which is the set of pairs \((x, \sigma)\) where \( x \) is an existing identity and \( \sigma \) is the current \( A \)-state vector of \( x \). \( \sigma \) will contain \( \text{NA} \)'s at places corresponding to inapplicable attributes. Now, in the domain, all values in \( \sigma \) will be known, whereas in the DB, some may be unknown. Whereas in the domain we distinguish \( \forall \) from \( \text{NA} \), in the DB we will have to add another type of null value, \( \perp \), representing "unknown." The UR model does not distinguish these
different types of null values.

To illustrate the difference in the treatment of null values in UR models and in ABSURD, we translate an example given by Maier et al. [1984] in ABSURD. The example DB state is represented by the three relation instances

<table>
<thead>
<tr>
<th>emp</th>
<th>child</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jones</td>
<td>Ann</td>
</tr>
<tr>
<td>Jones</td>
<td>Jim</td>
</tr>
<tr>
<td>Green</td>
<td>Sue</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>emp</th>
<th>dept</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jones</td>
<td>Shoes</td>
</tr>
<tr>
<td>Smith</td>
<td>Toys</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>dept</th>
<th>mgr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toys</td>
<td>Green</td>
</tr>
</tbody>
</table>

Table 1.

Taking the natural join and chasing the dependencies given by Maier et al. (not listed here) leads to the UR instance

<table>
<thead>
<tr>
<th>emp</th>
<th>child</th>
<th>dept</th>
<th>mgr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jones</td>
<td>Ann</td>
<td>Shoes</td>
<td>2</td>
</tr>
<tr>
<td>Jones</td>
<td>Jim</td>
<td>Shoes</td>
<td>2</td>
</tr>
<tr>
<td>Green</td>
<td>Sue</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>Jones</td>
<td></td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>Smith</td>
<td></td>
<td>9</td>
<td>Green</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>12</td>
<td>Green</td>
</tr>
</tbody>
</table>

Table 2.

To represent this example in ABSURD, we can either use our intuitions concerning the UoD represented by table 1 and represent those in an ABSURD model, or we can try to translate table 1 itself as faithfully into ABSURD as possible. For fairness of comparison, we choose the second option, even though we do not utilize the full expressive capability of ABSURD to describe the UoD. The following attribute axioms are then appropriate:

\[
\begin{align*}
\text{name} &= [\text{Emp} \cup \text{Child} \cup \text{Dept} \rightarrow \text{NAMES}] \\
\text{children} &= [\text{Emp} \rightarrow \mathcal{O}(\text{Child})] \\
\text{dept} &= [\text{Emp} \rightarrow \text{Dept}] \\
\text{mgr} &= [\text{Dept} \rightarrow \text{Emp}].
\end{align*}
\]

In a more accurate UoD abstraction, we would have introduced the natural kind Person and defined children as \([\text{Person} \rightarrow \mathcal{O}(\text{Person})]\). This would have necessitated a decision about the value of children (Ann), since Ann is a Person (who in table 1 is only recorded as a child). Figure 8 shows the generalization, aggregation and set hierarchies involved.

To translate table 2 into ABSURD, we are faced with the problem that Smith, who is an
employee, is recorded as having \( \bot \_9 \) children. It is not clear whether this means "unknown number of children" or "no children". Similar remarks hold for the \( \bot \_2 \) manager of the shoes department. Assuming that Smith has no children and shoes has no manager, we get table 3.

<table>
<thead>
<tr>
<th>identity</th>
<th>name</th>
<th>children</th>
<th>dept</th>
<th>mgr</th>
</tr>
</thead>
<tbody>
<tr>
<td>( e_1 )</td>
<td>Jones</td>
<td>( {p_1, p_2} )</td>
<td>( d_1 )</td>
<td>NA</td>
</tr>
<tr>
<td>( e_2 )</td>
<td>Green</td>
<td>( {p_3} )</td>
<td>( d_2 )</td>
<td>NA</td>
</tr>
<tr>
<td>( e_3 )</td>
<td>Smith</td>
<td>( \emptyset )</td>
<td>( d_2 )</td>
<td>NA</td>
</tr>
<tr>
<td>( p_1 )</td>
<td>Ann</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>( p_2 )</td>
<td>Jim</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>( p_3 )</td>
<td>Sue</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>( d_1 )</td>
<td>shoes</td>
<td>NA</td>
<td>NA</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( d_2 )</td>
<td>toys</td>
<td>NA</td>
<td>NA</td>
<td>( e_2 )</td>
</tr>
</tbody>
</table>

Table 3.

The left hand side of table 3 is \( \text{ext}_w(\text{em}) \). For example,

\[
\begin{align*}
\text{ext}_w(\text{Emp}) &= \{e_1, e_2, e_3\} \\
\text{ext}_w(\text{Child}) &= \{p_1, p_2, p_3\} \\
\text{ext}_w(\text{Dept}) &= \{d_1, d_2\}.
\end{align*}
\]

The right hand side of table 3 is the set \( \text{ST}_A(\text{ext}_w(\text{em}))_w \), the set of \( A \)-state vectors of
existing identities. The state of existing identities results when we project an $A$-vector on $type(x)$. Thus,

$$w(e_1) = (Jones, \{p_1, p_2\}, d_1) \text{ and } w(d_2) = (toys, e_2).$$

Note the following things about table 3.

3.1. Each tuple represents a meaningful domain object, because the identity of the object about which the attributes give information is represented. Compare this with the tuples in table 2, where no such subject of information is singled out (the attributes in table 2 occur in arbitrary order).

3.2. There is only one NA value in table 3, not distinguished by an index. This expresses the fact that there is only one kind of inapplicability. Much of the indexing of nulls in table 2 is thus superfluous.

3.3. Although the UR model is supposed to handle null values, it cannot distinguish the case where an attribute value is unknown from the case where it is positively known that it there is no component present. In ABSURD, there is no incomplete knowledge at the domain level and we can use $\emptyset$ to denote absence. In the DB, we can expect some of the definite values in table 3 to be replaced by $\perp$.

Some other differences between ABSURD and the UR approach are really difference between ABSURD and the relational approach in general and will be treated in the next subsection.

7.2.3.

A fundamental difference between the ABSURD model and relational models is that in relational models, an attribute is

1. a set of entities as well as
2. a role which these entities play (see Beeri & Korth [1982] for a clear statement of this view).

This leads to possible confusions when the entities "in an attribute" play different roles. For example, an Emp may play the roles of a project leader, chief engineer, father, and manager in the same database. If we would use the same name for the entity set Emp as for the appropriate attribute in Project, Working-group, Family and Dept, we would neglect to give a name to these different roles in different environments; but if we would give each attribute an intuitively satisfying name, we would neglect the fact that these attributes have the same underlying "domain." By distinguishing attributes from attribute ranges, these distinctions can be expressed.

The ABSURD concept of attribute has the advantage that several related assumptions can be distinguished, viz. the unique role, unique attribute name and unique attribute range assumptions. These assumptions say that

1. The functions in an attribute belong together in that they formalize the same role, independent of the identities to which they currently happen to apply.
2. Different attributes have different names and each attribute has the same name in each world, and that
3. Each attribute draws its values from the same set of possible identities in each world.

Separation of entity from role can only be done when we have a way of naming entities in a attributeless, neutral way. This is done by ABSURD. Use of identities has another advantage, that of no-duplication. Whereas within a relation instance key values are not duplicated, between relation instances they are, e.g. as referential keys. By using identities we avoid this at the conceptual level. The use of identities also allows us to express a further uniqueness condition independently of the current state of the world, viz.

4. Each identity has a unique name.

7.2.4.

Maier & Warren [1982] and Maier et al. [1982] introduce a concept of object in their UR model which differs from the ABSURD concept. They define an association as a subset of the universal set of attributes \( U \) which is nondecomposable in some sense. For each database scheme a set \( A \) of associations is defined. In their example with attributes class, instructor and student, the associations are \{class\}, \{class, instructor\} and \{class, instructor, student\} (note the identification of entity sets with attributes). The nondecomposable relationships expressed by these associations are

- there is a catalog of classes,
- an instructor is scheduled to teach a class, and
- a student is registered to take a class from an instructor.

For each database scheme a set \( O \) of objects is also defined. \( A \subseteq W \) of \( U \) is an object if the relation on \( W \), \( r(W) \) is the natural join of relations on associations,

\[
\bar{r}(W) = \bigwedge_{X \in A, x \subseteq W} r(X).
\]

A database scheme \( d(A, O) \) is a set of associations and a set of objects.

For an arbitrary set \( Z \) of attributes, the window \([Z]\) for \( Z \) on the database, or the connection for \( Z \) in the database, is

\[
[Z] = \bigcup_{w \in O, z \subseteq W} \Pi r(W).
\]

In ABSURD terms, Maier's associations and objects are natural types and the connection on an arbitrary set of attributes \( t \) is \( \pi_{A,t}([w^t(\text{ext}_w(t))]) \), the set of \( t \)-vectors of the current extension of \( t \). The difference between Maier's associations and his objects is that associations correspond to smallest natural types; objects are unions of types.

Maier et al. formulate containment conditions for associations, objects and connections.

The containment conditions are

1. For associations \( X, Y \in A : X \subseteq Y \Rightarrow \pi_X(r(Y)) \subseteq X \).
2. For objects \( V, W \in O : W \subseteq V \Rightarrow \Pi_W(\bar{r}(V)) \subseteq \bar{r}(W) \).
3. For the connections for \( X, Y \subseteq U : X \subseteq Y \Rightarrow \Pi_X([Y]) \subseteq [X] \).

2 and 3 follow from 1 and the definitions. Translating this into ABSURD, we replace \( X \subseteq U \) by \( t \subseteq A \) and the tuples of a relation on \( X \) by the identities in \( \text{ext}_w(t) \). All containment conditions then reduce to
\[ t_1 \subseteq t_2 \Leftrightarrow \text{ext}_w(t_2) \subseteq \text{ext}_w(t_1) \]

ABSURD thus allows us to express the containment conditions in a simpler and richer way, for theorems 4.1.8 and 5.4.5 give additional statements for natural types and dual statements for kinds and natural kinds.

7.3. Non first normal form relations

7.3.1. The need for non first normal form relations (N1NF relations) was recognized ten years ago by Makinouchi [1977] and Furtado & Kerschberg [1977] and, via the introduction of multivalued dependencies (MVD's), by Fagin [1977]. Research into MVD's has stayed squarely within the relational tradition and research into N1NF models has concentrated primarily on the extension of the relational model with repeating groups and the definition of relational operators and a normal form for relations with set-valued attributes. (Abiteboul & Bidoit [1984], Arisawa et al. [1983], Fischer & Furtado & Kerschberg [1977], Hull & Yap [1984], Jaeschke & Schek [1982], Kuper & Vardi [1984], Ozsoyoglu & Ozsoyoglu [1983], Ozsoyoglu & Yuan [1985], Schek & Scholl [1986], Thomas [1983]). The extensions to the relational model have alternatively been called "Non-first-normal-form relations," "formats," "quotient relations," "summary tables," "nested relations," and "relations with relation-valued attributes." Extensions of relational operators to relations with set-valued attributes are treated in Fischer & Thomas [1983], Jaeschke & Schek [1982], Furtado & Kerschberg [1977], Ozsoyoglu & Ozsoyoglu [1983], and Schek & Scholl [1986]. Normal forms for N1NF relations are defined by Arisawa et al. [1983] and Ozsoyoglu & Yuan [1985]. These papers also prove some results concerning the connection between MVD's and N1NF relations. Query evaluation for N1NF relations is discussed in Kuper & Vardi [1984].

A second strand of research, outside that of the relational tradition, is the study of complex objects in object-oriented databases (Bancilhon & Koshafian [1986], Khoshafian & Copeland [1986]). Complex objects are tuples or sets of primitive or complex objects and can thus be viewed as a reincarnation of tuples in a N1NF database.

A third strand of research stands in the logic programming tradition and tries to extend Prolog with set-valued variables (Beeri et al. [1987], Kuper [1987]). This research concentrates on giving a denotational and equivalent procedural semantics for logic programming languages with sets.

In this section, we compare ABSURD with Fagin's MVD's and with the LDL language defined by Beeri et al. [1987]. A comparison of the ABSURD object concept with that Khoshafian & Copeland [1986] was made in section 2.1.

7.3.2. As an example of normalization to 4NF with MVD's, Fagin [1977] gives an example database scheme of a university database. In the university UoD, students follow classes which are divided into sections. Different sections have different instructors and different meetings days and rooms. All sections in a class use the same texts. A student has several scores for exams.
for a particular section and class. A student has a name, a year (e.g. sophomore) and a major (e.g. Math). An instructor has a name, a salary and a rank. The universal scheme to be normalized is 

\[ U = \{ \text{CLASS, SECTION, STUDENT, MAJOR, EXAM, YEAR, INSTRUCTOR, RANK, SALARY, TEXT, DAY, ROOM} \}. \]

The MVD's recognized by Fagin are

\[ (\text{CLASS, SECTION}) \rightarrow+ \{ \text{STUDENT, MAJOR, EXAM, YEAR} \} \]

\[ (\text{CLASS, SECTION}) \rightarrow+ \{ \text{INSTRUCTOR, RANK, SALARY} \} \]

\[ (\text{CLASS, SECTION}) \rightarrow \text{TEXT} \]

\[ (\text{CLASS, SECTION}) \rightarrow \{ \text{DAY, ROOM} \} \]

\[ \text{CLASS} \rightarrow \text{TEXT} \]

\[ (\text{CLASS, SECTION, STUDENT}) \rightarrow \text{EXAM} \]

After normalization, Fagin comes up with the following 4NF database scheme:

\[ R_1 = (\text{CLASS, SECTION, STUDENT, EXAM}) \]

\[ R_2 = (\text{STUDENT, MAJOR, YEAR}) \]

\[ R_3 = (\text{INSTRUCTOR, RANK, SALARY}) \]

\[ R_4 = (\text{CLASS, SECTION, INSTRUCTOR}) \]

\[ R_5 = (\text{CLASS, TEXT}) \]

\[ R_6 = (\text{CLASS, SECTION, DAY, ROOM}). \]

Compare this with the attribute names in \( A \) and the aggregation graph in figure 9.

\[ A = \{ \text{name, sections, texts, instructor, students, sessions, salary, rank, major, year, day, room, score} \}. \]

In the graph we see set-valued attributes
sections = [Class → Ψ(Section)]
texts = [Class → Ψ(Text)]
students = [Section → Ψ(Student)]
sessions = \([\text{Section} \rightarrow \mathcal{P}(\text{Session})]\)

score = \([\text{Section} \times \text{Student} \rightarrow \mathcal{P}(\text{Score})]\)

The attribute students corresponds to the MVD (1) in that it expresses that there more than one students per section. It expresses the relevant real-world fact more precisely than 91), which carries also independent information, viz. that a class has sections and a student a major, exam, and year. If even more precision is desired, the set constructor \(\mathcal{P}^+(x)\) could be defined as \(\mathcal{P}(x) - \emptyset\) and students as

\[
\text{students} = \text{Section} \rightarrow \mathcal{P}^+ (\text{Student}).
\]

MVD (2) and (1) hide the information that a class has several sections. Once this is factored out by the attribute sections, (2) becomes superfluous. Similarly, (3) and (5) express both the information that a class has several texts used by all sections of the class. This is expressed more concisely by texts. The attribute sessions states more precisely what (4) attempts to say, viz. that a section has several sessions which can meet at different times and places. Finally, the essence of (6) is that a section-student combination can have several scores, is expressed by score.

This brief example illustrates the intimate connection between MVD’s and N1NF relations and at least suggests that the UoD structures expressed by MVD’s can be more naturally expressed in ABSURD.

7.3.3.

A recent paper by Beeri et al. [1987] reveals the need for sound axiomatic foundations for logic programming with sets. In the language defined by Beeri et al., LDL1, an extension of the Logic Based Database Language developed at MCC. There are two ways to define sets in LDL1, by enumeration, which is “the process of constructing a set from its elements,” and by grouping, which is “defining its elements by a property (i.e., a conjunction of predicates) that they satisfy.” (Beeri et al. p. 22) One would expect these two ways to be recognizable LDL1 variants of the two standard ways to define sets in ZF (or in other axiomatizations). Thus, in ZF a is defined by enumeration and b by grouping:

\[
a = \{x, y, z\}
\]

\[
b = \{x \mid \phi(x)\}.
\]

Translation of the LDL1 examples of enumeration and grouping shows that this is not so. An example of enumeration in LDL1 is

\[
\text{book\_deal}(\{X, Y, Z\}) \leftarrow \text{book}(X, Px), \text{book}(Y, Py), \text{book}(Z, Pz), Px + Py + Pz < 100.
\]

The predicate book\_deal is true of all sets of three books whose total price is less than 100. In LS, this is the set

\[
\text{book\_deal} = \{x \mid x \subseteq \text{Book} \land |x| = 3 \land \text{sum}(\text{price}[x]) < 100\}.
\]

We assume a function sum which, applied to a set of integers, yields the sum of the elements of its argument, and a natural kind Book to whose elements the attribute price is applicable. book\_deal is thus defined by the LDL1 equivalent of grouping, not enumeration.

Beeri et al. give the following example of grouping in LDL1.
part(P#, <Subpart#>) ← p(P#, Subpart).

This groups a relation instance of (PART#, SUBPART#) into an instance of a N1NF relation in which each part number occurs with the set of its subparts. This is not really an operation in $L_5$. Instead, the subparts of a part would be reachable by the attribute

$\text{subparts} = [\text{Part} \rightarrow \mathcal{B}(\text{Subpart})].$

The LDL1 program which uses the above rule to compute the total cost of all subparts is

\[
\begin{align*}
\text{part(P#, <Subpart#>)} & \leftarrow p(P#, \text{Subpart}). \\
\text{tc}({X}, C) & \leftarrow q(X, C). \\
\text{tc}({S}, C) & \leftarrow \text{part}(X, S), \text{tc}(S, C). \\
\text{tc}(S, C) & \leftarrow \text{partition}(S, S1, S2), \text{tc}(S1, C1), \text{tc}(S2, C2), +(C1, C2, C3). \\
\text{result}(P#, C) & \leftarrow \text{tc}({P#}, C).
\end{align*}
\]

This is expressed more concisely in $L_5$ as

\[
\text{sum}([\text{cost}\ [\text{subpart}[\text{Part}]]].
\]

Two remarks should be made at this point. First, LDL1's concept of grouping is apparently motivated by a desire to turn 1NF relations into N1NF relations. This attacks a problem created by the state of relational theory: The real world contains sets and set-valued attributes which can quite efficiently be implemented as repeating groups, and there is no need to take a detour through 1NF representations. All papers cited above about N1NF relations show how to turn 1NF concepts into N1NF concepts, i.e. they define the semantics of N1NF normal forms, operations and queries in terms of 1NF normal forms, operations and queries. This calls for a second quotation from Atzeni & Parker [1982]:

By assuming a purely relational model with no semantic structures other than data dependencies, one naturally encounters difficulties that would not arise in "real" databases. In making assumptions (such as the universal relation assumption) more for establishing certain results (such as the usefulness of 'decomposition' as a design methodology) than for their appropriateness in modelling data, one must realize that the results may not be useful to database engineers.

The ABSURD approach is to step back from current research and describe explicitly which structures we want to build database models of. The language for explicit description is an extension of $L_{ZF}$ and, as the examples in this chapter show, this language certainly allows us to state the essence of the relevant domain structures.

This brings us to the second remark. $\Sigma_5$ is not meant for implementation in a finite machine; if anything, the domain, as an abstraction of the UoD, is the implementation. If we can only describe the essence of the domain by dropping the demand of implementability, one can wonder whether we have chosen a useful level of abstraction. I hope the above examples, and the ones to follow in the next subsection, amply illustrate the usefulness of $L_5$ in the analyst's attempt to get a clear conceptual picture of the UoD (i.e., to construct the domain in an empirical fashion. How $L_5$ can be used in combination with an empirical method to derive
true general statements about the UoD will be investigated in a later report.)

I close this subsection with an example which shows why a conceptual tool like ZF, which almost literally is the size of the universe, is needed to crack an egg the size of the UoD. Beeri et al. give the following example of an LDL1 program which has no model.

\[ p(\langle X \rangle) \leftarrow p(X). \]

Shortcircuiting the explanation Beeri et al. give of the (non-)meaning of this program, it groups all elements of \( X \) and place the result as an element into \( X \). Obviously, this violates the axiom of regularity, for there is no \( E \)-minimal element of \( X \). We have

\[ \forall Y \in X[Y \cap X \neq \emptyset]. \]

According to Beeri et al., this "is reminiscent of the Russel-Whitehead paradoxes." (ibid., p. 25). If this is a problem, that means that the axiom of regularity is assumed to hold in models of LDL1 programs. This asks for a more careful elaboration of the relation between LDL1 programs and axiomatic set theory. The top-down approach followed in this report, extending the language of ZF with constants and using it to talk about possible worlds which sit in the universe of axiomatic set theory, starts on the right foot from the beginning and, in addition to having a sound formal basis, allows one to express the relevant domain structures in a precise and concise manner.

7.4. "Semantic" data models

7.4.1.

The word "semantic" is placed between quotes because it is hard to see what a nonsemantic model would be like. (See note 2 in the preface.) We use the word because it is the term used in the literature. In a handy overview and comparison, Urban & Delcambre [1986] evaluate a number of "semantic" data models on their static and their dynamic modelling capabilities according to an explicit list of criteria. As explained in the beginning of this chapter, these models take a level of abstraction between the DB and the domain and consequently Urban & Delcambre talk about the domain as well as the database. For example, talk of objects, classes, properties, classification etc. is clearly applicable to the domain (as well as the database), but talk of operations on the database, updates, internal identifiers, procedural attachment etc. is applicable to the database only. In the following evaluation, we only look at criteria applicable to the domain, and regard them as criteria for the evaluation of the structures commonly found in domains. We treat the criteria for static structures only, for these are the topic of this report.

The criteria are divided into six groups, which have the following headings.

1. The identification of objects.
2. The classification of objects.
3. The aggregation of objects.
5. Association of objects.
Each of the data models reviewed by Urban & Delcambre, RM/T, TAXIS, SDM, SHM+, and the Event Model, is judged by them on the availability of structures falling under these headings. We devote a short section to the criteria in each of these groups. In order to show that the ABSURD concepts can provide a conceptual foundation for the data models discussed by Urban & Delcambre, we quote and discuss their definitions as well as discussing their results. Unless otherwise stated, all quotations are from Urban & Delcambre [1986].

7.4.2. Objects

"Objects represent both the concrete and abstract entities of interest in the application domain." Objects are obviously supported by our model, with a slight but significant change in terminology. The "application domain" is our UoD. In ABSURD, objects represent concrete or conceptual entities in the UoD instead of concrete or abstract entities. Use of the word conceptual emphasises the importance of the interest which the people living in the UoD have in the represented entities. The objects themselves are abstract entities in the domain, and represent entities in the UoD. All data models discussed by Urban & Delcambre support some sort of object construct.

"Internal identifiers uniquely represent the existence of each distinct object in the database, allowing objects to be directly interrelated without relying on external identifiers as foreign keys." Dropping the words "internal" and "external," this applies to ABSURD. Most models evaluated by Urban & Delcambre support some notion of identity.

7.4.3. Classes

"A class describes a collection of objects with common properties." This notion is supported by all data models and by ours as well. As a critical remark it may be noted that none of the models make very clear what it is to have common properties. What is it that two ships of different type, owned by different companies, commanded by a different captain and located in different parts of the globe have in common? In this report, what they have in common is explicated as the applicability of attributes. It is this common applicability which makes it possible to draw up the list of differences in the last sentence but one. A correct notion of property is important for theoretical as well as practical reasons. For theoretical reasons, because not every property determines a set (the property \( x \in x \) being a case in point). For practical reasons, because the system designer needs some clues in deciding what to look for when dissecting the UoD.

Some of the specialities offered by some of the data models are:

**Cardinality constraints.** The Event Model allows the user to define constraints on the cardinality of the number of instances in a class. The static part of such a constraint can be expressed quite easily as a conjunct in \( IC \), though formulation of the dynamic part of it must wait till a forthcoming report. The number of existing objects in a world is \( |ext_w(em)| \) and the number of existing persons is \( |Persons_w| \).

There is some sloppiness in the notion of class multiplicity as it is discussed (very briefly) by King & McLeod [1984]. They talk about a class as "representing" the existence of a set of objects. In ABSURD terms, what is meant is that the extension of a kind has a certain cardinality, i.e. \( k_w = n \) for an \( n \in \mathbb{N} \).
Class attributes. SDM supports the notion of properties of "a class as a whole." For example, \textit{average-\textit{age}} is a property of a class of persons. This is represented naturally in our data model, because all identities are sets, albeit that some have their elements outside \( S \). Thus, we can easily define an attribute

\[
\text{average-age} = [(\text{Person}) \rightarrow \text{nat}]
\]

subject to the integrity constraint

\[
\forall x \in \mathcal{F(\text{Person})}[\text{average-age}_w(x) = \text{sum}(\text{age-w}[x]) / |x|],
\]

where \textit{sum} computes the sum of a set of natural numbers.

The treatment of class properties in SDM is criticized by Urban & Delcambre because class properties cannot be inherited by subclasses in SDM. What is meant, presumably, is that if \( k_0 \) is any set of identities, that any subset \( k_1 \) of \( k_0 \) should be at least of the same type, i.e. all attributes applicable to elements of \( k_0 \) should be applicable to the elements of \( k_1 \). This is necessarily so in our model. Theorem 4.1.4.6 says that

\[
k_1 \subseteq k_0 \Rightarrow \text{type}(k_0) \subseteq \text{type}(k_1).
\]

Recall that this is valid for all \( k \subseteq S \), also those which are sets of subsets of \( S \). An immediate consequence, for example, is that

\[
k_1 \subseteq k_2 \Rightarrow \mathcal{P}(k_1) \subseteq \mathcal{P}(k_2) \Rightarrow \text{type}(\mathcal{P}(k_2)) \subseteq \text{type}(\mathcal{P}(k_1)).
\]

An analogous statement can be derived for extensions \( k_w \).

In sum, the specialities offered by some data models are all present in our model. Integration of all of them in one framework requires a more coherent framework than these data models offer. In particular, the following types of sets need be distinguished. Each of the distinctions made is violated by at least one data model.

1. A \textit{set} is a collection of objects.
2. A \textit{type} is a set of attributes.
3. A \textit{natural kind} is the set of all possible objects to which all attributes in a given type apply.
4. The \textit{extension} of a natural kind in a world is the set of existing objects in \( w \) of that kind.
   The extension of a type is the set of existing objects of that type.
5. A \textit{set identity} is an identity which has all its elements in \( S \). A \textit{set attribute} is an attribute whose value has all its elements in \( S \).
6. A \textit{class} is a set of objects, i.e. (identity, state) pairs.

7.4.4. Aggregation

Atomic objects like social security numbers or names can be aggregated into aggregate objects like persons. Urban & Delcambre look at two aspects of aggregation, the nature of the objects aggregated and the support for "semantic relativism." We first treat the list of special features concerning the nature of the objects aggregated.

Set-valued attributes. "SDM and the Event Model both directly support multi-valued properties, while RM/T, TAXIS, and SHM+ rely on single-valued properties only." Set-valued attributes are a central element of our model.
Exhausting a value set. In SDM, RM/T, and the Event Model one can specify that the attribute members of elements of Project must "exhaust its value set," i.e. that each employee must be a member of a project. In $L_5$, this would be expressed as the integrity constraint

$$\text{Emp}_w \subseteq \text{members}_w(\text{Project}_w).$$

Note that we have resolved an ambiguity in the concept of value exhaustion in that we require existing employees to be members of existing projects.

Non-overlapping attribute values. This facility is defined by Hammer & McLeod [1981] (p. 363) as follows:

"...the values of the attribute for two different entities have no entities in common, that is, each member of the value class of the attribute is used at most once."

This can mean two things. Expressed in $L_5$, the first possible meaning of the definition is that $a$ has non-overlapping values if

$$\forall x, y \in \text{ext}_w(\{a\}) [x \neq y \Rightarrow a_w(x) \cap a_w(y) = \emptyset].$$

This constraint can be strengthened to a statement about all possible values for $a$,

$$\forall x, y \in \text{dom}(a) \forall f \in a [x \neq y \Rightarrow f(x) \cap f(y) = \emptyset].$$

The second possible meaning of the definition of non-overlapping values is that $a_w$ is a 1-1 function, which is to say that it is a key: $a = [k_1 \rightarrow k_2]$.

The example they give suggest that this is what they mean:

"For example, Engines of SHIPS is specified as having "no overlap in values," which means that any engine can be only in one ship."

Constant vs. time-varying attributes. The ABSURD possible worlds framework allows us to distinguish operations, which deliver the same result in every possible world, from attributes, whose result depends upon the current world. Attributes can be divided in constant attributes, which cannot be changed once an object to which the attribute is applicable is in existence, and variable attributes, which may change once an object to which the attribute applies has been instantiated. Constant attributes could have received another value at instantiation time, but cannot be changed after instantiation time (See section 3.3). None of the models discussed here distinguishes constant attributes from operations.

Null values. This is a moot point, if only because the meaning of a null value is usually left open to interpretation by the user of the database. We allow only one "null value," $\emptyset$, with the meaning absent. name($x$) = $\emptyset$ means that $x$ has no name and members($y$) = $\emptyset$ means that $y$ has no members. Most models support the use of a "null value," but the semantics of these values has not been worked out satisfactorily. Of the 14 different types of null values distinguished by ANSI [1975] (cited by Atzeni & Parker [1982]), we have formalized inapplicability and absence. Of the remaining ones, some with dynamic connotations (such as "not yet existed") will be formalized in a forthcoming report about domain dynamics. Others, like "not known" belong to the DB, not the domain.

Urban & Delcambre also discuss virtual properties, which are "properties that are defined through procedural attachment or through operations on other properties in the database." This is clearly a database concept. In the domain, the difference between stored and computed
values does not have a meaning.

The second major aspect of aggregation, next to the nature of the objects aggregated, is semantic relativism. Semantic relativism is defined by Brodie [1984] as "the ability to view and manipulate data in the way most appropriate for the viewer." Put this way, it belongs to the database, not the domain. The domain theory describes the world in terms of which all user views must be defined. In general, a finite database will store information which represents not one, but several domains and this ambiguity will be transferred to the user views defined on the database. All post-relational data models support some kind of semantic relativism.

7.4.5. Generalization

"Generalization is a form of abstraction in which objects are viewed as higher-level generic objects, suppressing the differences between objects and emphasizing their common properties.... Specialization is the reverse of generalization." "All of the models support generalization, but they differ in the manner in which generalization hierarchies are formed." Not only this, they do not have an explicit theory of generalization. In ABSURD, generalization is primarily a relation between types and secondarily between natural kinds. If \( k_1 \subseteq k_2 \), then \( k_1 \) is a specialization of \( k_2 \) iff

\[ \exists t_1, t_2 \left[ k_1 = \text{kind}(t_1) \land k_2 = \text{kind}(t_2) \land t_2 \subseteq t_1 \right]. \]

There are three headings under which Urban & Delcambre evaluate generalization,

1. Inheritance of properties,
2. Restriction of inherited attribute values, and
3. Formation of the generalization hierarchy.

**Inheritance of properties.** All models support inheritance of attribute applicability. There are other types of inheritance, though, such as inheritance of default values for attributes. Because "property" can mean attribute as well as attribute value, it is often not clear what exactly is supported by the different systems. Inheritance of attribute values comes again in two specialities, strict value inheritance, in which all instances of a class share a certain attribute value, and default inheritance properly so called, which is the instantiation of an object of a class with default values for certain attributes unless these values are overridden. Default logic is still in its infancy and is not treated here. Strict value inheritance is a special case of attribute restriction, treated below.

**Attribute restriction.** TAXIS, SDM and the Event Model provide a facility for restricting the inherited attribute values. This can be represented in two ways in our model. One is to define a predicate on a natural class, e.g. \( \{ x \in \text{Person} \mid \text{sex}(x) = \text{male} \} \). This defines a subset of Person. We have not provided a way to name this set (unless it is equal to a identity, in which case it is named by \( s \)) and we do not regard this set necessarily as a specialization of Person, because the set of attributes applicable to it may be the same as type(Person). The other way to restrict attribute values is to take a specialization, e.g. \( \text{Secr} \subseteq \text{Emp} \), and define a subkind of this kind, for example \( \{ x \in \text{Secr} \mid \text{age}(x) < 30 \} \). In both cases, there are two aspects to be distinguished.

First, the introduction of names for defined sets is a matter of the user interface, not of the domain. The sets defined by attribute restrictions are part of the domain, but their names, if
not in CON, are part of the user interface.

Second, assuming that all sets we may possibly be interested in have a name in the domain theory, we may ask what the logic of specialization by attribute restriction is. There is an interesting metaphysical question connected with this, which is whether we get a more detailed specialization hierarchy if we add specialization by attribute restriction to specialization by attribute inheritance. If \( k \) is a natural kind, is it necessarily so that the kind \( k' = \{ x \in k \mid \phi(x) \} \) (if it is a set) is a natural kind in our sense? That is, can we find a non-trivial attribute which is applicable precisely to the elements of \( k' \) and to no other identities? If so, attribute restriction adds nothing to attribute inheritance.

An interesting speculation to be investigated in the future is whether identities that share a common generic process form a natural kind. Can the concept of natural kind be sharpened by allowing predicates on kinds which single out a set of identities which in some sense "behave similarly?" Common life cycles are an important element of the biological concept of natural kinds (see Mayr [....]). But the most important element in the biological concept of natural kind is common descent. The formalizability of these matters in the context of database domain modelling is a matter worthy of future investigation.

**Formation of the generalization hierarchy.** This is really a matter of the user interface. Generalizations are not "formed," they simply exist in the domain. However, names may be given to certain sets, and this is a process in the user interface. But the four ways in Urban & Delcambre's summary in which generalizations can be "formed" can be viewed as four ways to describe sets, and as such they will be discussed. It will be surprising if we cannot account for them, for ZF is the one of the strongest ways to describe sets which is consistent.

1. **Attribute-defined** subclasses pose no problem in our model, as is seen by the example given above.
2. **User-defined** subclasses are not part of the domain.
3. **Set-operator defined** subclasses are obviously definable by us.
4. **Existence-defined** subclasses in SDM are sets of objects which are the value of a specified attribute. For example, the set DANGEROUS – CAPTAINS is defined as the set of OFFICERS which are a value of the attribute involved – captain of INCIDENTS (Hammer & McLeod [1981], p. 359). 17 In \( L_5 \), this set is described as involved – captain \( w [\text{Incidents}_w] \).

**Multiple inheritance.** This is supported by all models except the Event model. It is central in our model (4.2).

**Mutual exclusion of subclasses.** This is supported by SHM + and (optionally) by RM/T and SDM. It is also a central element in our model (4.2).

17. This is an interesting case of the influence on the state of the UoD by the presence of an information system. The concept of dangerous captain is reified, if not ossified, and made available to users of the information system, who will apply their intuitive understanding of it to the captain involved, who is thereby ostracized and may be petrified.
7.4.6. Association

"Association is a form of abstraction in which objects (possibly heterogenous in nature) are viewed as members of a higher-level set object." It is not clear what this adds to the concept of generalization as defined above. In ABSURD terms, an "association" is simply a non-natural kind. An example of the use of a non-natural kind is

\[ \text{members}: \text{Project} \rightarrow \mathcal{P}((\text{Secr} \cup \text{Eng}) \cap \text{Perm}). \]

\( \mathcal{A}_1 \cup \mathcal{A}_2 \) is not a natural kind (4.2.6).

If anything, Urban & Delcambre's discussion of association shows that "semantic modelling" lacks conceptual foundations. The current report is an attempt to provide such foundations. As can be seen from this comparison, the concepts which have been developed are not new, but have a clarity which facilitate the description of the issues involved and the comparison of the solutions proposed.

Association is supported by RM/T, SDM and SHM+, but only RM/T supports association of heterogenous objects as defined above for members.

7.4.7. Metaclasses

"A metaclass is a class in which the instances themselves are classes." In our terminology, a metaclass is a set whose elements are subsets of \( S \). Just as for the concept of association, the concept of metaclass adds nothing to the set concept as we adopted it from ZF. Rather, it detracts from it, for the elements of the elements of a metaclass cannot be sets themselves. The use of sets as identities is, not surprisingly, a very powerful idea.

The term "metaclass" is perhaps a bit mystifying as compared to the ZF concept of a set. Usually, "meta" is used to indicate "aboutness." For example, English is the metalanguage for \( L_5 \), for we use it to talk about those languages. Similarly, the data dictionary (DD) of DBMS is a metadatabase, for its contents describe the database, i.e. are about it. The data in a DD are metadata. We may structure the DD in any way, but it is folklore to give it the same structure as the DB which it represents. It will then contain objects which have a kind, type and class, but it would go to far (and against the decision to give it the same structure as the DB) to use a different terminology for the DD and talk of metakinds, metatypes and metaclasses.

Metaclasses are supported by TAXIS and SDM. The two headings under which Urban & Delcambre looked at metaclasses are

1. Formation of metaclasses and
2. Generalization of metaclasses.

Concerning the first heading the same remarks can be made as for the formation of the generalization hierarchy. The two types of formation discussed by Urban & Delcambre are attribute-defined and user-specified; in our model the full range of descriptional facilities is available for metaclasses as for sets, for metaclasses are sets.

Only TAXIS supports generalization of metaclasses. This, too, is described effortlessly in our model.
7.4.8.

It thus turns out that the use of ZF allows the precise expression of advanced domain structures. Realization of those structures in a database is, of course, a wholly different matter but this is not the immediate aim of the present research. (But see appendix 4, where the models of the domain axiomatization may be used to define the structure of global database identifiers.)

I believe that the above comparison shows that the study of domain structures independently of time- and space limitations is the proper way to get to grips with the conceptual perplexities of database domain modelling. The static structures studied here are very complex. The state spaces of physical systems have a simpler structure in that generalization, aggregation and sets do not occur in such a complex way. Explication of these structure in $L_5$ has the advantage that decisions must be made at the stage where they belong, during the analysis of the UoD, and not when it may be too late, during implementation. At implementation time we may discover that we implicitly have made decisions which turn out to be wrong or improper to the UoD. If these had been made during analysis of the UoD, at least they would have been made consciously, and the implementation options could have been described with a clearer understanding of the UoD.
Chapter 8
Conclusion

8.1. Summary
I have formalized the domain structures of classification, aggregation, grouping and specialization into a coherent framework which is interpreted in a set-theoretical universe. The concepts and formal tools used are drawn from four worlds, that of database theory, set theory, object-oriented programming and logic. The binding element is the concept of identity. In ABSURD, an identity is a set and attributes are sets of functions on identities. Identities can be classified into natural kinds, which in this report are just identities with the same aggregation structure but in a forthcoming report are isomorphic identities executing the same generic process. The fact that objects have an internal state which can be changed by events and that the effect of an event depends upon the state of the world, distinguishes natural kinds from ADT’s.

The set, aggregation and generalization structures are axiomatized in $\Sigma_5$, for which $\mathbb{N}$ effectively functions as a model (appendix 4). This creates a direct link with the implementation, notably with the structure of global database identifiers. Each model for the axiomatization can be turned into a set of possible worlds which do or do not satisfy the static integrity constraints. When a world satisfies the static constraints, it is called admissible.

The major simplifying assumptions made in this and the following reports are:
1. The merging of two or more entities into one is not described.
2. The splitting of an entity into two or more is not described.
3. A change of applicability of attributes is not described.
4. Specialization is by attribute addition only.

Even with these limitations, the ABSURD model proved to be a powerful tool to conceptually clarify the proposals for "semantic" models and non first normal form models.

8.2. Future research
Much work still has to be done to show that the ABSURD approach is viable. On the side of the UoD, ABSURD concepts should be tried out in an information analysis to see how much conceptual help they give in the formation of clear abstractions from the UoD. On the side of the DB, finiteness should be taken into account and it should be shown how an ABSURD model can be implemented in a finite database. In the domain itself, the static possible worlds structure developed in this report should be extended with a dynamic component to specify van Fraassen’s "laws of succession and interaction."

To round out this picture, at the level of the user interface a number of interesting practical and philosophical questions crop up. Should all identities be visible to the user? Should they be pictured in linear or two-dimensional format? On a more principled level, what is the effect of a precise specification of domain dynamics on the UoD? As is coming to be realized in the data modelling community, an information system not only contains statements of fact about the UoD, it also contains speech acts which influence the UoD (Lehtinen & Lyytinen [1986]).
References

Abbreviations:

CACM  Communications of the ACM
JACM  Journal of the ACM
OOPSLA Proceedings of the Conference on Object-Oriented Programming Systems and Languages
PODS  Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems
TODS  Transactions on Database Systems
VLDB  Proceedings of the International Conference on Very Large database Systems

Abiteboul S., N. Bidoit [1984]
"Non First Normal Form relations to represent Hierarchically Organized Data," third PODS, Waterloo, Ontario, 191-200.

Albano A., L. Cardelli [1985]

Albano A., M.E. Occhiuto, R. Orsini [1986]
Galileo Reference manual, VAX™/UNIX™ Version 1.0, Dipartimento di informatica, Università di Pisa, Italy.

ANSI [1975]

Arisawa H., K. Moriya, T. Miura [1983]
"Operations and the Properties of Non-First-Form Relational Databases," 9th VLDB, 197-204.


Atzeni P., D. S. Parker [1982]
Bancilhon F., S. Khoshafian [1986]
"A Calculus for Complex Objects," 5th PODS.

Barwise J. [1975]
Admissable sets and Structures, Springer.

Beeri C., H.F. Korth [1982]

Beeri, C., S. Naqvi, R. ramakrishnan, O. shmueli, S. Tsur [1987]

Benacerraf P. H. Putnam (eds.) [1964]

Bernays P. [1935]

Biskup J., H.H. Brüggemann [1983]

"Generalization/Specialization as a Basis for Software Specification," in Brodie et al. [1984], 87-114.

Brodie M.L. [1984]

Brodie M., J. Mylopoulos, J. Schmidt (eds.) [1984]
On Conceptual Modelling, Springer-Verlag, New York.

Brodie M.L., D. Ridjanovic [1984]

Brodie M.L., E. Silva [1982]
Codd E.F. [1979]

Date C.J. [1983]

Enderton H.B. [1972]

Fagin R. [1977]

Fischer P.C., S.J. Thomas [1983]

Fraassen B.C. van [1970]

Fraenkel A.A. [1968]

Furtado A.L., L. Kerschberg [1977]

Gamut L.T.F. [1982]

Griethuysen, J.J. van (ed.) [1982]

Hall P., J. Owlett, S. Todd [1976]
Hammer M., D. McLeod [1981]

Hintikka J. [1969]

Hollingdale S.H., G.C. Tootill [1975]

Hughes G.E., M.J. Cresswell [1968]
An Introduction to Modal Logic, Methuen, London.

Hull, R., C.K. Yap [1984]

Jacobs B.E. [1982]

Jaeschke G., H.-J. Sebek [1982]
"Remarks on the Algebra of Non First Normal Form relations," 1st PODS, 124-137.

Kent W. [1978]
Data and Reality, North-Holland.

Kent W. [1979]

Kent W. [1980]
"Splitting the Conceptual Schema," Proc. of the 6th Int. conf. on Very large Database, Montreal, 10-14.

Kerschberg L. (ed.) [1984]
Expert Database Systems, Benjamin/Cummings.

Khoshafian S.N., G.P. Copeland [1986]
Kripke S. [1971]

Kripke S. [1980]
*Naming and Necessity*, revised and enlarged edition, Basil Blackwell.

King R. [1984]
"A Database Management System Based on an Object-Oriented Model," in Kerschberg [1984], 443-468.

King R., D. McLeod [1984]

King R., D. McLeod [1985]

Kuper G.M. [1987]

Kuper G.M., M.Y. Vardi [1984]

Kutschera F. von [1976]
*Einführung in die Intensionale Semantik*, de Gruyter, Berlin.

Langefors B. [1966]
*Theoretical Analysis of Information Systems*, Studentlitteratur.

Lehtinen E., K. Lyytinen [1986]

Levesque H. J. [1984]

Maier D., D.S. Warren [1982]
"Specifying Connections for a Universal Relation," *Proc. of the ACM-SIGMOD Int. Conf. on Management of Data*, 1-7

"Toward Logical Data Independence: A Relational Query Language without Relations," *Proc. of the ACM-SIGMOD Int. Conf. on Management of Data*, 51-60.

Maier D., D. Rozenshtein, D.S. Warren [1983]

Makinouchi A. [1977]

Mayr E. [1982]

Ozsoyoglu Z.M., G. Ozsoyoglu [1983]

Ozsoyoglu Z.M., L.-Y. Yuan [1985]

Maier D., J.D. Ullman, M.Y. Vardi [1984]

Mylopoulos J., P.A. Bernstein, H.T. Wong [1980]

Nagel T. [1986]
*The View from Nowhere*, Oxford Un. Press.

Olle T.W., H.G. Sol, A.A. Verrijn-Stuart (eds.) [1982]
Quine W.V.O. [1977]


Reiter R. [1984]


Schek H.-J., M.H. Scholl [1986]


Schwartz S.P. (ed.) [1977]

Naming, Necessity, and Natural Kinds, Cornell UP.

Searle J.R. [1969]


Searle J.R. [1984]


Searle J.R. & D. Vanderveken [1985]


Smith J.M., D.C.P. Smith [1977a]

"Database Abstractions: Aggregation," CACM Vol. 20, No. 6 (June 1977), 405-413.

Smith J.M., D.C.P. Smith [1977b]


Takeuti G., W.M. Zaring [1971]


Ullman J.D. [1982]


Ullman J.D. [1987]

Urban S.D., L.M.L. Delambre [1986]


Wieringa R.J. [1987]


Wieringa R.J. [forthcoming]


Winograd T., F. Flores [1986]

### Appendix 1
**List of notations**

<table>
<thead>
<tr>
<th>metavariables over the set of wff's</th>
<th>metavariables over individual variables (sets)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\phi, \psi, \eta)</td>
<td>(x, y, z, \ldots)</td>
</tr>
<tr>
<td>(A, B, \ldots)</td>
<td>metavariables over the individual variables and class symbols</td>
</tr>
<tr>
<td>(c_0, c_1, \ldots)</td>
<td>metavariables over (CON)</td>
</tr>
<tr>
<td>(a_0, a_1, \ldots)</td>
<td>metavariables over (CON_A)</td>
</tr>
<tr>
<td>(t_0, t_1, \ldots)</td>
<td>metavariables over attributes</td>
</tr>
<tr>
<td>(\ell_0, \ell_1, \ldots)</td>
<td>metavariables over types (subsets of (A))</td>
</tr>
<tr>
<td>(k_0, k_1, \ldots)</td>
<td>metavariables over kinds (subsets of (S))</td>
</tr>
<tr>
<td>(k_0, \kappa_1, \ldots)</td>
<td>metavariables over natural kinds (subsets of (S))</td>
</tr>
<tr>
<td>(\kappa_0, \kappa_1, \ldots)</td>
<td>metavariables over (CON_{NK})</td>
</tr>
<tr>
<td>(w_0, w_1, \ldots)</td>
<td>metavariables over worlds</td>
</tr>
<tr>
<td>(o_0, o_1, \ldots)</td>
<td>metavariables over objects (((s_i, o)))</td>
</tr>
<tr>
<td>(\sigma_0, \sigma_1, \ldots)</td>
<td>metavariables over states</td>
</tr>
<tr>
<td>(s_0, s_1, \ldots)</td>
<td>identity names (surrogate names)</td>
</tr>
</tbody>
</table>

Occasionally, \(t\) is used as metavariable over terms.

Metavariables are variables in the metalanguage (English). In the metalanguage, they can be quantified over (e.g. "for all \(c\) in \(CON\)"). As a symbol for a formula in the object language, \(\forall x[\phi(x)]\) is well-formed but \(\forall c \in CON[\phi(x)]\) is ill-formed.

Use of the metavariables over individual variables of \(L_S\) allows one to drop appropriate conditions, e.g.

\[\forall \kappa[\phi(\kappa)]\]

is an abbreviation of

\[\forall x[x \subseteq S \land x = kind(type(x)) \Rightarrow \phi(x)]\]

and

\[\exists \kappa[\phi(\kappa)]\]

of

\[\exists x[x \subseteq S \land x = kind(type(x)) \land \phi(x)].\]
Appendix 2
Zermelo-Fraenkel set theory

A.2.1.
An axiomatization of ZF is presented in Takeuti & Zaring [1971]. This appendix summarizes
the main points, borrowing some logical definitions from Enderton [1972].
We call the language of ZF $L_{ZF}$. The primitive symbols used in $L_{ZF}$ are

**Individual variables:**
There are infinitely many variables. These are not actually given, and the letters
$x, y, z, ..., $
are used as metavariables over the collection of individual variables.

**Predicate symbol:** $\in$

**logical symbols:** $\neg, \land, \lor, \Rightarrow, \Leftrightarrow, \forall, \exists$

**Auxiliary symbols:** $( ), [ ]$.

The letters
$\phi, \psi, \eta$
are metavariables over the set of well-formed formula's (wff's). For example, the formula
$\forall x [\phi(x)]$
is an expression in the metalanguage (English), from which a formula in $L_{ZF}$ may be gotten by
replacing $\phi$ by a string of the symbols listed above and $x$ by an actual variable of $L_{ZF}$. We
keep the convention that in $\phi(x)$ at least $x$ occurs as free variable. If we fix the individual vari­
ables of $L_{ZF}$ as
$x, y, z,$
then the result of the replacement might be the wff
$\forall x [\neg x \in x].$
A wff is a symbol constructed out of the primitive symbols of $L_{ZF}$ according to only the
following rules.
1. If $x$ and $y$ are individual variables, then $x \in y$ is a wff.
2. If $\phi$ and $\psi$ are wff's then $\neg \phi, \phi \lor \psi, \phi \land \psi, \phi \Rightarrow \psi$, and $\phi \Leftrightarrow \psi$ are wff's.
3. If $\phi$ is a wff and $x$ an individual variable then $\forall x [\phi]$ and $\exists x [\phi]$ are wff's.
If $x$ is free in $\phi$ we will often write $\phi(x)$ for $\phi$. We also abbreviate
$\forall x [x \in a \Rightarrow \phi(x)]$ by $\forall x \in a [\phi(x)]$
and
$\exists x [x \in a \land \phi(x)]$ by $\exists x \in a [\phi(x)]$.

A.2.2.
The logical axioms used by Takeuti & Zaring are

1. \( \phi \Rightarrow (\psi \Rightarrow \phi) \).
2. \( (\phi \Rightarrow (\psi \Rightarrow \eta)) \Rightarrow ((\phi \Rightarrow \psi) \Rightarrow (\phi \Rightarrow \eta)) \).
3. \( (\neg \phi \Rightarrow \neg \psi) \Rightarrow (\psi \Rightarrow \phi) \).
4. \( \forall x (\phi \Rightarrow \psi) \Rightarrow (\phi \Rightarrow \forall x \psi) \) if \( x \) is not free in \( \phi \).
5. \( \forall x \phi(x) \Rightarrow \phi(a) \) where \( x \) has no free occurrence in a well formed part of \( \phi \) of the form \( \forall a \psi \).

The inference rules are

1. From \( \phi \) and \( \phi \Rightarrow \psi \) infer \( \psi \).
2. From \( \phi \) infer \( \forall x \phi \).

If \( \Sigma \) is a set of sentences containing the logical axioms, then by \( \text{Ctx}(\Sigma) \) we mean the set of sentences which logically follow from \( \Sigma \). For example, if \( \Sigma_{\text{FOL}} \) is the set of logical axioms, then \( \text{Ctx}(\Sigma_{\text{FOL}}) \) is the set of sentences derivable from the logical axioms and if \( \Sigma_{\text{ZF}} \) is the set of axioms of ZF, then \( \text{Ctx}(\Sigma_{\text{FOL}} \cup \Sigma_{\text{ZF}}) \) is the set of all theorems of ZF.

A.2.3.

A major concern of axiomatic set theory is the avoidance of the paradoxes of naive set theory. In ZF, this is done by distinguishing \textit{sets} from \textit{classes}. The \textit{class symbol}

\[ \{x \mid \phi(x)\} \]

is used to denote the class of all \( x \) such that \( \phi(x) \). The letters

\[ A, B, \ldots \]

are used as metavariables over the individual variables and class symbols.

Every set is a class, but not every class is a set. A class which is not a set is called a \textit{proper class}. Formally, the predicate \( \mathcal{M} \) (expressing that its argument is a set) is defined by

\[ \mathcal{M}(A) := \exists x [x = A] \]

A well-known example of a proper class is the Russell class

\[ \{x \mid x \in x\} \]

The class of all sets is called \( \mathcal{V} \). In this report, we are only concerned with subsets of \( \mathcal{V} \). The word "class" is also used in a different way, to indicate certain sets of objects which have an internal state. Context will make clear in which sense the word is used.

To be able to talk about proper classes in \( L_{ZF} \), the concept of \textit{wff} is extended so that in addition to \textit{wff}’s certain formulas in which the symbol

\[ \{x \mid \phi(x)\} \]

occurs are well-formed as well. A \textit{wff in the wider sense} can be constructed from \textit{wff}’s in the original sense by using only the following rules.

1. If \( t_0 \) and \( t_1 \) are terms, then \( t_0 \in t_1 \) is a \textit{wff in the wider sense}. 
2. If \( \phi \) and \( \psi \) are wff's in the wider sense and \( a \) and \( b \) are individual variables, then
\[
\begin{align*}
& a \in \{ x \mid \phi(x) \}, \\
& \{ x \mid \phi(x) \} \in b, \text{ and} \\
& \{ x \mid \phi(x) \} \in \{ x \mid \psi(x) \}
\end{align*}
\]
are wff's in the wider sense.

3. If \( \phi \) and \( \psi \) are wff's in the wider sense then \( \neg \phi, \phi \lor \psi, \phi \land \psi, \phi \Rightarrow \psi \), and \( \phi \Leftrightarrow \psi \) are wff's in the wider sense.

4. If \( \phi \) is a wff in the wider sense and \( x \) an individual variable then \( \forall x [\phi] \) and \( \exists x [\phi] \) are wff's in the wider sense.

Every wff in the wider sense is an abbreviation of a wff in the original sense. Takeuti and Zarling define give rules for translating a wff in the wider sense to a unique wff in the original sense. As an example, if \( \phi \) and \( \psi \) are wff's in the original sense, then
\[
\{ x \mid \phi(x) \} \in \{ x \mid \psi(x) \}
\]
can be translated into
\[
\exists x [\psi(x) \land \forall y [y \in x \Rightarrow \phi(y)]]
\]

A.2.4.

\( L_{ZF} \) has only one predicate symbol, \( \in \). The following predicates can be defined in terms of \( \in \). These predicates are introduced by definitions, but are used as if they were primitive predicates and as if their definitions were added as axioms to \( \Sigma_{ZF} \). \( x, y \) and \( f \) are (metavariables over the) variables of \( L_{ZF} \).

\[
\begin{align*}
& \mathcal{U}(A), \\
& x = y, \\
& x \subseteq y, \\
& x \subset y, \\
& f: x \to y, \\
& f: x \to^{1-1} y, \\
& f: x \to_{onto} y, \\
& f: x \to^{1-1}_{onto} y.
\end{align*}
\]

A.2.5.

A symbol of the form \( \{ x \mid \phi(x) \} \) is called a *class symbol* and denotes the class of all entities which satisfy \( \phi(x) \). All of the classes defined in this report are sets. In particular, Takeuti & Zaring define the following sets. \( x \) and \( y \) are sets.

\[
\emptyset, \langle x, y \rangle \text{ (The ordered pair } x, y) \text{.}
\]
\( x \cap y \)
\( x \cup y \)
\( \mathcal{P}(x) \)
\( x - y \) (The difference between two sets.)
\( x \times y \) (Cartesian product.)
\[ f(x) \quad \prod_{x \in a} f(x) \]
\( f[x] \) (the image of \( x \) under \( f \). Only used if \( f \) is a function, but defined for every set \( f \).)
\( \text{dom}(f) \) (Will only be used for functions, but is defined for every set.)
\( \text{range}(f) \) (Only used for functions.)
\( x \circ y \) (Only used for functions.)
\( \in \) and arithmetical operations on natural numbers. (We use the symbol \( \in \N \) instead of the customary \( \omega \) of set theory.)

In addition, we assume the following sets to have been defined.
\( \mathcal{P}^+(x) \) (non-empty subsets of \( x \).)
\( \mathcal{F}(x) \) (Finite subsets of \( x \).)
\( \mathcal{F}^+(x) \) (Non-empty finite subsets of \( x \).)
\( \text{STRINGS}(a) = \bigcup_{n \in \N} \prod_{x \in n} (f(x)). \) (strings of 0 or more elements of \( a \).)
\( x \bullet y \) (concatenation of finite strings \( x, y \in \text{STRINGS}(a) \).)
\( |x| \) (length of a finite string \( x \in \text{STRINGS}(a) \).)
\( \leq_a \) (lexicographical ordering of strings over \( a \), where \( a \) is a well-ordered set.)
\( \mathbb{Q}, \mathbb{Z} \) and \( \mathbb{R} \), and arithmetical operations on these sets.

It is clear that we could also have defined matrices and matrix operations and other mathematical objects and operations, if necessary.

As an example of the translation of expressions in which these abbreviations occur into wff's of \( L_{ZF} \), \( x \cap y \) is the class \( \{z \mid z \in x \land z \in y\} \), so that the wff in the wider sense
\( a \in x \cap y \)
can be translated into the wff in the wider sense
\( a \in \{z \mid z \in x \land z \in y\} \)
and then into the wff
\( a \in x \land a \in y. \)

\textbf{A.2.6.}
The axioms in \( \Sigma_{ZF} \) are
1. Extensionality.
\( \forall a \forall x \forall y[x = y \land x \in a \Rightarrow y \in a]. \)
2. **Pairing.**
\[ \forall a \forall b. \mathcal{M}(\{a, b\}). \]

3. **Unions.**
\[ \forall a. \mathcal{M}(\bigcup(a)), \text{ where } \bigcup(a) \text{ is the union of the elements of } a. \]

4. **Powers.**
\[ \forall a. \mathcal{M}(\mathcal{P}(a)). \]

5. **Schema of replacement.**
\[ \forall a [ \forall u \forall v \forall w [\phi(u, v) \land \phi(u, w) \Rightarrow v = w] \Rightarrow \exists y \forall b [\exists b \Rightarrow \exists x [x \in a \land \phi(x, y)]]]. \]
(Intuitively, this says that functions map sets onto sets.)

6. **Regularity.**
\[ \forall a [a \neq \emptyset \Rightarrow \exists x [x \in a \land x \cap a = \emptyset]]. \] (This prevents the Russell paradox.)

7. **Infinity.**
\[ \mathcal{M}(\mathbb{N}). \]

We do not use the axiom of choice,
\[ \forall a \exists f \forall x \in a [x \neq \emptyset \Rightarrow f(x) \in x]. \]
(This axiom is equivalent to the statement that every set can be well ordered. \(a\) is well-ordered by \(R\) if there is an \(R\)-minimal element of \(a\) and every two elements of \(a\) are pairwise compatible.)

So far, we have given the syntax and derivation rules of a formal language and some logical (\(\Sigma_{FOL}\)) and nonlogical (\(\Sigma_{ZF}\)) axioms. We abbreviate \(\Sigma_{FOL} \cup \Sigma_{ZF}\) by ZF. In the rest of this report we extend \(L_{ZF}\) with constants naming objects in the domain and extend the axiom set ZF with axioms describing the domain. In other words, the domain is a structure for the extended language and is a model for the extended axiom set axioms. As a preliminary, we first define what it is to be a model of ZF.

### A.2.7. Definition
A **structure** for a first-order language \(L\) is a pair \([D, I]\) in which \(D\) is a set and \(I\) an interpretation function such that

1. \(I(c) \in D\) for each constant symbol \(c\);
2. \(I(f) : D^n \rightarrow D\) for each \(n\)-ary function symbol \(f\);
3. \(I(P) \subseteq D^n\) for each \(n\)-place predicate symbol \(P\).

Applied to \(L_{ZF}\), the situation is very simple because we have no constants or function symbols and only one predicate constant, \(\in\).

### A.2.8. Definition
\([D, I]\) is a structure for \(L_{ZF}\) if

1. \(D \subseteq V\), where \(V\) is the class of all sets, and
2. \(I(\in) \subseteq D \times D\).
[D, I] is a standard structure for $L_{ZF}$ if

$I(\in) = \in$.

It is called a standard transitive structure if

$x \in D \Rightarrow x \subseteq D$. □

We only consider standard transitive structures for $L_{ZF}$.

At first sight, the interpretation of the formal language of set theory in a structure described by the formal language of set theory may seem strange, but this procedure is not different from that followed for other languages. Each n-ary function symbol ($n = 0, 1, \ldots$) is assigned a function $D^n \rightarrow D$ (which is simply a constant when $n = 0$) and each n-place predicate is assigned a subset of $D^n$. What is peculiar about this procedure is that the language being interpreted is the language in which the model is described. In the interpretation of an arbitrary formal language $L$, we use the language of set theory to talk about a model of $L$. If $L$ is the language of set theory, we use $L$ to talk about a model of $L$. The symbols $\subseteq, \times$ and $\in$ used to define $[D, I]$ above, are the same symbols as those defined in $L_{ZF}$. This shows that $L_{ZF}$ is about as basic as we can get in reasoning about models in general and the domain of a DB in particular and that we have to bootstrap our way from axiomatic set theory to the theory of models of set theory.

The consequence of this bootstrap procedure is that truth of a wff in a structure is definable in $L_{ZF}$ itself.

A.2.9. Definition

Let $[D, I]$ be a standard structure for $L_{ZF}$. The symbol $\models$, to be read as "satisfies," is defined as a predicate in $L_{ZF}$ as follows. (:= is to be read as "is defined by.")

1. $[D, I] \models x \in y := x \in D \land y \in D \land x \in y$.
2. $[D, I] \models \neg \phi := \neg ([D, I] \models \phi)$.
3. $[D, I] \models \phi \land \psi := ([D, I] \models \phi) \land ([D, I] \models \psi)$.
4. $[D, I] \models \forall x [\phi(x)] := \forall x ([D, I] \models \phi(x))$. □

In other words, formula $\phi$ is true in $[D, I]$ iff a certain other formula is derivable in ZF. This other formula restricts the variables to $D$. The following definition and theorem make this precise.

A.2.10. Definition

Let $[D, I]$ be a standard structure for $L_{ZF}$. Then the wff $\phi^D$ is defined as follows.

1. $[x \in y]^D := x \in y$.
2. $[\neg \phi]^D := \neg \phi^D$.
3. $[\phi \land \psi]^D := \phi^D \land \psi^D$.
4. $[\forall x [\phi(x)]]^D := \forall x \in D [\phi^D(x)]$. □
In words, $\phi^D$ is $\phi$ with quantification restricted to $D$.

A.2.11. Theorem

Let $[D, I]$ be a standard structure.

1. If $\phi$ is closed then
   \[ [D, I] \models \phi^D. \]

2. If all free variables occurring in $\phi$ are among $x_1, \ldots, x_n$, then
   \[ x_1 \in D \land \ldots x_n \in D \Rightarrow [[D, I] \models \phi] \Rightarrow \phi^D]. \]

Proof.

See Takeuti & Zaring, theorem 12.5. $\square$

Truth of a closed formula $\phi$ in $[D, I]$ is therefore equivalent to derivability of $\phi^D$, where $\phi^D$ differs from $\phi$ only in that quantification is restricted to $D$ instead of to $V$. If $\phi$ has free variables, then the same statement must be qualified by demanding that $x \in D$ for all free variables $x$ of $\phi$. 
Appendix 3

The example domain axiomatization in $L_S(\text{CON})$

Constants of $L_S(\text{CON})$:

$CON = CON_A \cup CON_NK \cup \{NA, A, NK, em\}$

$CON_A = \{\text{ext}, \text{emp}\#, \text{salary}, \text{name}, \text{licence}\#, \text{typing-speed}, \text{skill}, \text{contr}\#, \text{#years}, \text{project}\#,$

$\text{project-type}, \text{members}, \text{task}\}$

$CON_NK = \{S, \text{Emp}, \text{Trucker}, \text{Ldtrucker}, \text{Sectr}, \text{Eng}, \text{Temp}, \text{Perm}, \text{Project}\}$

Predicates:

$\in$.

Defined sets:

$\text{EMP##, NAMES, LICENCE##, INT-LICENCE##, SPEEDS, SKILLS, CONTRACT##, PROJECT##, PROJECT-TYPES, TASKS.}$

Axioms:

$\Sigma_S = \Sigma_{FOL} \cup \Sigma_I \cup \Sigma_A \cup \Sigma_{NK} \cup \Sigma_{SP}$.

Logical axioms:

$\Sigma_{FOL} = \{
\phi \Rightarrow (\psi \Rightarrow \phi),
(\phi \Rightarrow (\psi \Rightarrow \eta)) \Rightarrow ((\phi \Rightarrow \psi) \Rightarrow (\phi \Rightarrow \eta)),
(\neg \phi \Rightarrow \neg \psi) \Rightarrow (\psi \Rightarrow \phi),
\forall x (\phi \Rightarrow \psi) \Rightarrow (\phi \Rightarrow \forall x \psi),
\forall x \phi(x) \Rightarrow \phi(t)\}$

Identity axioms

$\Sigma_I = \{
\exists S [N \rightarrow \text{onto} S],
\emptyset \in S,
\text{em} \in S\}$.

Attribute axioms:

$\Sigma_A = \{NA \in S,$

$\forall a \in A [a = \text{emp}\# \lor \ldots \lor a = \text{task}],
\text{ext} = [\text{Ext} \rightarrow \mathcal{F}(S)],
\text{emp}\# = [\text{Emp} \rightarrow^{1-1} \text{EMP##}],
\text{salary} = [\text{Emp} \rightarrow N^+],
\text{name} = \text{Emp} \cup \text{Project} \rightarrow \text{NAMES}],
\text{licence}\# = [\text{Trucker} \rightarrow^{1-1} \text{LICENCE##}],
\text{int-licence}\# = [\text{Ldtrucker} \rightarrow^{1-1} \text{INT-LICENCE##}].$
typing-speed = [Secr \to SPEEDS],
skill = [Eng \to SKILLS],
contr# = [Contract \to^{1-1} CONTRACT#],
#years = [Perm \to \mathbb{N}]
project# = [Project \to^{1-1} PROJECT#],
project-type = [Project \to PROJECT-TYPES],
members = [Project \to \mathcal{F}(Emp \times TASK)].

Natural kind axioms:
\[ \Sigma_{NK} = \{ NK \subseteq \mathcal{P}(S), \]
\[ \{ \text{Emp, ..., Project-member} \} \subseteq NK, \]
\[ \text{kind(type(Emp))} = \text{Emp}, \]
\[ \text{kind(type(Project))} = \text{Project} \} . \]

Specialization axioms:
\[ \Sigma_{SP} = \{ S = \text{Emp} \cup \text{Project} \cup \text{Project-member}, \]
\[ \text{Emp} \cap \text{Project} = \emptyset, \text{Emp} \cap \text{Project-member} = \emptyset, \text{Project} \cap \text{Project-member} = \emptyset, \]
\[ \text{Emp} = \text{Trucker} \cup \text{Secr} \cup \text{Eng}, \]
\[ \text{Trucker} \cap \text{Secr} = \emptyset, \text{Trucker} \cap \text{Eng} = \emptyset, \text{Secr} \cap \text{Eng} = \emptyset, \]
\[ \text{Ldrucker} \subseteq \text{Trucker}, \]
\[ \text{Emp} = \text{Temp} \cup \text{Perm}, \]
\[ \text{Temp} \cap \text{Perm} = \emptyset \} . \]
Appendix 4
A model for the example domain axiomatization

To define a model for $\Sigma$, we must define the interpretation function in the structure $[V, I]$ such that $I(\in) = \in$ and $I(c) \in D$ for each $c \in \text{CON}$. A simple model for the example domain theory is as follows. $\mathcal{F}(x)$ is the set of finite subsets of $x$.

1. $I(S) = \mathbb{N} \cup (\mathbb{N} \times \mathbb{N}) \cup \mathcal{F}(\mathbb{N})$.
2. $I(em) = -2$.
3. $I(NA) = -1$.
4. $I(Emp) = \{2n + 1 \mid n \in \mathbb{N}\}$.
   $I(Trucker) = \{6n + 1 \mid n \in \mathbb{N}\}$.
   $I(Secr) = \{6n + 3 \mid n \in \mathbb{N}\}$.
   $I(Eng) = \{6n + 5 \mid n \in \mathbb{N}\}$.
   $I(Task) = \{12n + i \mid n \in \mathbb{N}, i = 1, 3, 5\}$.
   $I(Perm) = \{24n + i \mid n \in \mathbb{N}, i = 1, 7\}$.
   $I(Ldrucker) = \{24n + i \mid n \in \mathbb{N}, i = 1, 7\}$.

It can easily be seen that these sets satisfy the specialization axioms. Some examples of attribute definitions are

5. $I(emp#) = \{\{2n + 1 \mid n \in \mathbb{N}\} \rightarrow \mathcal{F}(\{2n + 1 \mid n \in \mathbb{N}\})\}$.
   $I(members) = \{\{2n + 2 \mid n \in \mathbb{N}\} \rightarrow \mathcal{F}(\{2n + 1 \mid n \in \mathbb{N}\} \times I(TASK))\}$.

These interpretations respect the axioms in $\Sigma$. If it is wished that $EMP##$ or other defined sets be disjoint from other sets, then this should be added to $\Sigma_{SP}$.

The natural kind axioms are satisfied as well. For example, in this model $Emp$ is precisely the set of numbers in the intersection of the domain of all attributes in $\text{type}(Emp)$, so $Emp = \text{kind} \text{type}(Emp))$. If $Emp$ would have been given a different interpretation, for example $\{4n + 1 \mid n \in \mathbb{N}\}$ (with the interpretation of the subkinds adjusted accordingly), then $Emp$ would not have been a natural kind.

It is clear that this model is not minimal. For example, $S$ could have been interpreted as $\mathbb{N} \cup \{2n + 1 \mid n \in \mathbb{N}\} \times \{2n + 2 \mid n \in \mathbb{N}\} \cup \mathcal{F}(\{2n + 1 \mid n \in \mathbb{N}\})$. What is needed is not so much a minimal model but one which is convenient in that the implementation can easily check of which kinds an identity is by looking at the form of the number which implements the identity.

Any implementation can only dispose of a finite subset of $\mathbb{N}$. The interpretation function for a finite model will therefore not be 1-1. This opens up a number of issues which will have to await future research, in particular what is and what is not a "convenient" model for the implementation. Since all these models are models of the same domain axiomatization, it is irrelevant as far as domain modelling is concerned which model is chosen. It is an interesting phenomenon that the choice of this high level of abstraction allows such fundamental implementation issues to come out as the choice of global database identifiers.