Object identifiers, keys, and surrogates — object identifiers revisited

Roel Wieringa & Wiebren de Jonge
Faculty of Mathematics and Computer Science,
Vrije Universiteit
De Boelelaan 1081a, 1081 HV, Amsterdam
The Netherlands
Email: roelw@cs.vu.nl, wiebren@cs.vu.nl

Appeared in *Theory and Practice of Object Systems*, 1(2), 1995, pages 101–114

Abstract

Sound naming schemes for objects are crucial in many parts of computer science, such as database modeling, database implementation, distributed and federated databases, and networked and distributed operating systems. Over the past 20 years, physical pointers, keys, surrogates and object identifiers have been used as naming schemes in database systems and elsewhere. However, there are some persistent confusions about the nature, applicability and limits of these schemes. In this paper we give a detailed comparison of three naming schemes, viz. object identifiers, internal identifiers (often called surrogates) and keys. We discuss several ways in which identification schemes can be implemented, and show what the theoretical and practical limits of applicability of identification schemes are, independently from how they are implemented. In particular, we discuss problems with the recognition and authentication of identifiers. If the identified objects are persons, an additional problem is that object identification may conflict with privacy demands; for this case, we indicate a way in which identification can be combined with privacy protection.

1 Introduction

Object identification is a crucial issue in many branches of computer science, ranging from operating systems to computer networks, database systems and programming languages. The importance of good object identification schemes becomes even greater with the advent of networked and distributed operating systems, and information systems that communicate with each other over EDI networks. Actually, object identification is vital for any administration, computerized or not. Much of the business of administrations and of applying computers in the real world is about manipulating identifications of objects.

Due to this importance, many different naming schemes have been devised, such as physical pointers, keys, surrogates and object identifiers. However, there is considerable confusion about the nature, applicability and limits of these schemes. For example, in many cases, physical pointers, keys or surrogates are used as object identifiers. Although many use object identifiers and agree that they are a good thing, few say what they actually mean by it. Those who do, tend to say incompatible things. This is a problem for theory as well as practice. Confusion about what object identifiers are and how they can be used is a hindrance to practical and theoretical progress. In addition, inappropriate uses of naming schemes can cause serious flaws in system design. Examples are given later.

In this paper, we give a precise definition of object identification schemes, compare this with the older concepts of key and surrogate, discuss ways to implement identification schemes, and show what the limits of the applicability of identification schemes are. Our goal is conceptual analysis and consolidation of what has been written about object identification. A result of this analysis is, we hope, a better understanding of the nature and limits of object identifiers, that provides a firm theoretical basis for the practical use of object identifiers.

In section 2, we define the concepts of naming scheme, oid scheme and oid (= object identifier). In section 3, we compare oids with keys and internal

identifiers (called surrogates by some authors). In section 4 we give some examples and, more interestingly, non-examples of oid schemes. Section 5 discusses possible mechanisms for assigning oids to objects. It also discusses the problems of recognizing the presence of an oid and of authenticating oids, and gives oid borrowing as a way to alleviate these problems somewhat. Section 6 defines the information transfer problem, shows a way to avoid the problem by a suitable use of object identifiers, and in addition shows how this can be combined with the demand for privacy protection. In section 7, we compare our view with the view of object identification in a number of other papers. Section 8 concludes the paper.

2 Object identifiers

Basically, an oid is a proper name of an object such that the connection between the oid and the object is one-one and fixed. This simple idea, when put into practice, contains many pitfalls, that can only be avoided by a careful analysis of the naming relation between the oid and the object. For example, if the identity of a person is represented by a number, then we can represent this number in 1's complement or 2's complement notation, with 16-bit words or 32-bit words, as a character string in ASCII or EBCDIC, etc. What does this do to the one-one relation between the oid and the object? Are there many different (but equivalent) oids of one object? What happens if a machine changes its internal representation; is this a change in oid? But were oids not supposed to be fixed? To see how this puzzle can be resolved, suppose one machine in a distributed database system uses the number 123 in 2's complement notation as the oid of a person, and another machine in the same distributed system uses the number 123 in 1's complement notation as the oid of a person, then these two *different* bitstrings represent the *same* oid and therefore represent the same object. So apparently, to determine whether two symbol occurrences represent the same oid, we must know which notation system must be used to determine the value represented by the symbol. By convention, we will subscript symbol occurrences with the notation system to be used for interpreting them.

To take another example, suppose two different database systems both use the number 123 as the oid of two different persons, and that one person has oid 456 in one database system and oid 789 in the other database system. Again, what does this do to the one-one relationship between oids and objects? Because the database systems are different, we cannot conclude from equality of oids used by different systems that the represented person is the same, nor can we conclude from difference of oids that the represented persons are different. We can draw such conclusions only if the two database systems are known to use the same naming scheme. So apparently, oid occurrences must also be subscripted by the naming scheme according to which oids are assigned to objects. (The difference between naming scheme and notation system is explained below.)

As a final motivating example, some object-oriented systems use the class name as part of an oid. This seems convenient to represent class change: just change the class name in an oid. What happens in this case to the fixed nature of the relation between an oid and a named object? And even if an object never changes class, it may have multiple classifications. Does this mean that the object has as many different oids as it has classes? And does this not destroy the fixed and one-one relation between oids and objects?

To sort out these and other problems, we start from the observation that, at rock-bottom level, the identity of objects is represented in administrations by means of *symbol occurrences*. These correspond to abstract *symbols*, that in turn represent *values*. It is these values that constitute oids, not so much the symbols or symbol occurrences. In the next subsection, we define different equality relations for these levels of abstraction. The distinction between symbol occurrences, symbols and values and the three equality relations are needed to lay a firm foundation for the construction of naming schemes. Naming schemes are defined as relations between values and objects, and oid schemes can then be defined as a special kind of naming scheme.

2.1 Symbols and equality

We assume a set V of abstract entities, which we call **values**. The set V itself is called a **value space**. The elements of V are abstract entities, which we hold to be unobservable and unchanging. All of us have seen many representations of the number 2, but have we ever seen the number 2 itself? Similarly, all of us have seen many representations of the character A, but have we ever seen the (abstract) character A itself? We do not assume that the elements of V exist, nor

that they do not exist; for our purposes, the question whether values really exist or not is irrelevant.

Next, we assume a set S of **symbols**. The set S itself is called a **symbol space**. Each symbol is a type that may have many instances, also called **symbol occurrences**. A symbol occurrence is an observable part of the world, such as a spoken word, a written word, an icon on a screen, or a word in computer memory. The concept of existence is applicable to symbol occurrences. For example, "a" is an occurrence of a symbol of which many occurrences exist on the sheet of paper that you are reading now. Without loss of generality, in the rest of this paper, we assume fixed sets S and V.

We assume that for each symbol, there is an agreed observation procedure that tells us whether a given occurrence is an occurrence of that symbol. It is not important here what this procedure is, as long as it assigns symbols unambiguously to occurrences. Thus, for the simplicity of our presentation, we require that different symbols have disjoint sets of occurrences. In this paper, we assume that all occurrences of one symbol are isomorphic. Thus, "E" and "E" are two occurrences of the same symbol, but "*E*" is an occurrence of a different symbol.

The text you are reading now[1], like any other text, only consists of symbol occurrences. Because we want to be able to talk about symbol occurrences, symbols as well as values, we need a textual convention to indicate when a symbol occurrence in this text denotes itself, a symbol, or a value. In the rest of this section, we will use " " to denote symbol occurrences, ' ' to denote symbols, and use symbol occurrences (without quotes) to denote values. For example, "E" is an occurrence of the symbol 'E', which has as value the character E.

A **notation system** is a partial function $n : S \rightarrow V$ (see figure 1). The domain of n, written $dom(n)$, is the set of symbols for which n is defined. For any $s \in dom(n)$, we call the value $n(s)$ the **denotation** of s.

What the meaning (value) of a symbol is depends upon the notation system used to interpret it. If context is not sufficient to disambiguate the intended meaning of a symbol or symbol occurrence, we will subscript it with the notation system in terms of which to interpret it. For example, in the decimal notation system (*dec*), "1" and "12" are occurrences of symbols '1' and '12', that denote the numbers 1_{dec} and 12_{dec}, respectively. In general one symbol can have many corresponding values. For example, the symbol 'E' in the hexadecimal notation system (*hex*) has as value E_{hex}, which is equal to 14_{dec}. In the alphabetic character notation system (*char*), 'E' has the character E_{char} as its value.

The sets V and S come with identity relations, which have extension $\{\langle v, v\rangle \mid v \in V\}$ and $\{\langle s, s\rangle \mid s \in S\}$, i.e. values and symbols are only identical to themselves. The same is true of symbol occurrences. We can now define several equality relations (figure 2).

- Any value, symbol or symbol occurrence is only **equal** to itself. We denote equality in this strict sense by $=$.

For example, "E"$_{char} \neq$ "E"$_{char}$, because these two occurrences are not identical to each other, but only to themselves. Howewer, we have 'E'$_{char}$ = 'E'$_{char}$ and $\mathrm{E}_{char} = \mathrm{E}_{char}$. Moreover, we have 'E'$_{hex}$ = 'E'$_{char}$ because we have two occurrences of the same symbol, and $\mathrm{E}_{hex} \neq \mathrm{E}_{char}$, because these occurrences are interpreted as different values.

Because symbols denote values, we can define a second type of relation, which we call equivalence and which is based on the denotation relation:

- Two (subscripted) symbols are **equivalent** (denoted by $\equiv$) if and only if they denote the same value.

For example, using the subscript *dec* to represent the decimal notation system, the subscript *bin* to represent the binary notation system and the subscript *hex* to represent the hexadecimal notation system, '14'$_{dec}$, '1110'$_{bin}$ and 'E'$_{hex}$ are non-identical symbols, but they are equivalent (they denote the same value): $14_{dec} \equiv 1110_{bin} \equiv \mathrm{E}_{hex}$.

We also apply the equivalence relation to symbol occurrences:

- Two (subscripted) occurrences of symbols are **equivalent** (also denoted by $\equiv$) if and only if their symbols are equivalent.

So "E"$_{char} \equiv$ "E"$_{char}$ because 'E'$_{char} \equiv$ 'E'$_{char}$, and "E"$_{hex} \not\equiv$ "E"$_{char}$ because 'E'$_{hex} \not\equiv$ 'E'$_{char}$. In addition, we define a third relation for symbol occurrences, called congruence.

- Two (subscripted) occurrences of symbols are **congruent** (written as $\cong$) if they are occurrences of the same symbol.

[1] Strictly spoken, the text *occurrence* you are reading now ...

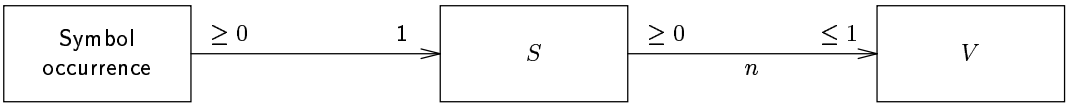

Figure 1: Relationships between symbol occurrences, symbols, and values. The cardinality constraint "≥ 0" means "any natural number, including 0", and "≤ 1" means "0 or 1". So each symbol has 0 or more occurrences and each value has zero or more symbols, related to it by a notation system n. In addition, each symbol occurrence is an occurrence of exactly one symbol, and a notation system n assigns at most one value to a symbol, which is its denotation.

	Strictly equal $=$	Equivalent $\equiv$	Congruent $\cong$
	Exactly the same	Same value	Same symbol
Values	X		
Symbols	X	X	
Occurrences	X	X	X

Figure 2: Three meanings of the general concept of equality. The entries of the table indicate applicability.

	Equal $=$	Equivalent $\equiv$	Congruent $\cong$
	Exactly the same	Same value	Same symbol
Values	$\mathrm{E}_{char} = \mathrm{E}_{char}$ $\mathrm{E}_{hex} \neq \mathrm{E}_{char}$ $\mathrm{E}_{hex} = 14_{dec}$ $\mathrm{E}_{char} = E_{char}$		
Symbols	$\text{'E'}_{char} = \text{'E'}_{char}$ $\text{'E'}_{hex} = \text{'E'}_{char}$ $\text{'E'}_{hex} \neq \text{'14'}_{dec}$ $\text{'E'}_{char} \neq \text{'}E\text{'}_{char}$	$\text{'E'}_{char} \equiv \text{'E'}_{char}$ $\text{'E'}_{hex} \not\equiv \text{'E'}_{char}$ $\text{'E'}_{hex} \equiv \text{'14'}_{dec}$ $\text{'E'}_{char} \equiv \text{'}E\text{'}_{char}$	
Occurrences	$\text{“E”}_{char} \neq \text{“E”}_{char}$ $\text{“E”}_{hex} \neq \text{“E”}_{char}$ $\text{“E”}_{hex} \neq \text{“14”}_{dec}$ $\text{“E”}_{char} \neq \text{“}E\text{”}_{char}$	$\text{“E”}_{char} \equiv \text{“E”}_{char}$ $\text{“E”}_{hex} \not\equiv \text{“E”}_{char}$ $\text{“E”}_{hex} \equiv \text{“14”}_{dec}$ $\text{“E”}_{char} \equiv \text{“}E\text{”}_{char}$	$\text{“E”}_{char} \cong \text{“E”}_{char}$ $\text{“E”}_{hex} \cong \text{“E”}_{char}$ $\text{“E”}_{hex} \ncong \text{“14”}_{dec}$ $\text{“E”}_{char} \ncong \text{“}E\text{”}_{char}$

Figure 3: Examples of equality, equivalence and congruence. By E_{char} we mean the value denoted by the symbol 'E' in the alphabetic character notation system, i.e. $char(\text{'E'})$. By E_{hex} we mean the value denoted by the symbol 'E' in the hexadecimal notation system, i.e. $hex(\text{'E'})$.

For example, "E"$_{hex} \cong$ "E"$_{char}$, because we have two instances of the same symbol, and "E"$_{hex} \ncong$ "E"$_{char}$, because the two symbol occurrences are are assumed to represent different symbols. Figure 2 summarizes the applicability of the three kinds of equality relations and figure 3 gives a number of examples. Note that the subscripts are only relevant for the interpretation of the first row (value equality) and the second column (equivalence). The examples show a regularity, viz. that symbol equivalence follows value equality, and that congruence of symbol occurrences follows equality of symbols. This agrees with the definition of the three kinds of equality relations.

2.2 Naming schemes

Object identifiers (oids) are a special kind of proper names used for denoting real world objects. To make us independent of the notation systems used, we use *values* rather than symbols as proper name. For example, if 353764558 is an oid of a person, then we can represent this by a symbol written on paper or on a screen, a bit string in one's complement notation, a bar code, etc. and still say that occurrences of these different symbols represent the same oid (i.e. are equivalent in our terminology).

To make the concept of oid more precise, we assume a symbol space S, a name space V, and a notation system $n : S \to V$ as before. In addition, we assume a set O of *all possible* objects. The set O is called the **object space**. The set O contains all *possible* objects that might possibly be named, so without loss of generality, we can assume that O does not change.

A **naming relation** is a subset of $V \times O$. The elements of a naming relation are assignments of proper names to objects. Naming relations may change during a state transition of the world. To make this explicit, let Σ be the set of all possible states of the world. Then we define a **naming scheme** N to be a function

$$N : \Sigma \to \wp(V \times O),$$

where $\wp(V \times O)$ is the powerset of $V \times O$. The function N assigns to every possible state of the world $\sigma \in \Sigma$ a naming relation $N_\sigma \subseteq V \times O$. In database terms, N_σ is a many-many relationship. Here, it is also convenient to view N_σ as a function $V \to \wp(O)$. The situation is represented in figure 4.

We can now get from a symbol to an object in two steps, in which a notation system is composed with a naming relation:

$$S \stackrel{n}{\to} V \stackrel{N_\sigma}{\to} \wp(O).$$

Because in general, there may be several naming schemes applicable to a value, in order to know which object(s) a symbol denotes, we must subscript it with a notation system, a naming scheme and a state, e.g. as in

$$s_{n,N,\sigma}.$$

We usually suppress these subscripts when they are clear from the context of a symbol occurrence.

To give an example of this, let *bar* be a notation system in which natural numbers are represented by bar codes, and let *card* be a naming scheme in which natural numbers are used as names for library users. Using obvious type definitions, in each state σ of the world we have

$$BARCODE \stackrel{bar}{\to} NATURAL \stackrel{card_\sigma}{\to} \wp(USER).$$

Then if b is a bar code, $card_\sigma(bar(b))$ is the set of library users identified by the number $bar(b)$ in state σ of the world. (In this example, the intention is of course that $card_\sigma(n)$ is a set of at most one element.) However, suppose for the sake of the example that the same bar code is also used to identify furniture. Let us call this naming relation $furn$:

$$BARCODE \stackrel{bar}{\to} NATURAL \stackrel{furn_\sigma}{\to} \wp(FURNITURE).$$

Then $furn_\sigma(bar(b))$ is the furniture identified by $bar(b)$ in state σ. In both cases, we can of course switch notation systems without changing anything in the naming scheme. For example, we may represent the number $bar(b)$ by a bit string b' in computer memory, using a notation system *bit*:

$$BITSTRING \stackrel{bit}{\to} NATURAL.$$

We would then have $bit(b') = bar(b)$, so that both symbols could be used for the same naming purposes.

We define the domain and range of a naming relation as follows:

$$dom(N_\sigma) = \{v \mid \exists o \in O \ : \ \langle v, o\rangle \in N_\sigma\},$$
$$range(N_\sigma) = \{o \mid \exists v \in V \ : \ \langle v, o\rangle \in N_\sigma\}.$$

In any state σ of the world, there may be names in V that are not assigned by N_σ to any object in O, and there may be objects in O that are not assigned a name by N_σ. Examples of nameless objects are products such as pins, nails and screws, where the overhead of giving them a name is not worth the trouble.

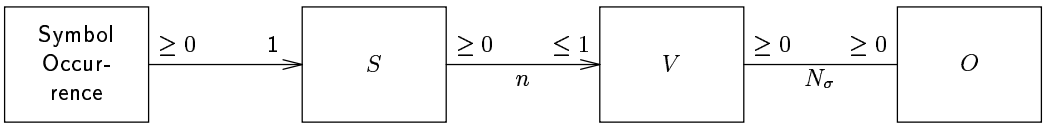

Figure 4: Relationships between symbol occurrences, symbols, values and objects. A naming relation N_σ assigns to each value a set of zero or more objects that it names, and to each object a set of zero or more values that are its names.

In administrations, however, every explicitly represented object will be named (otherwise it could not be represented explicitly)[2].

Finally, note that naming and existence can be independent of each other. There may be named objects that do not exist in σ. For example, we may give a car a name (e.g. a serial number) before it is produced and we may continue using the proper name of a car, even after the car has been demolished.

2.3 Oid schemes

In the following we will identify the requirements that a naming scheme must satisfy in order to be called an **object identification scheme** (oid scheme). If a naming scheme N is an object identification scheme, then in each state σ, the elements of N_σ are called **oid assignments** and the elements of $dom(N_\sigma)$ are called **oids**.

At least the following two requirements should be satisfied by an oid scheme.

- **Singular reference**. A naming scheme N satisfies the singular reference requirement if in each possible state σ of the world, each proper name in $dom(N_\sigma)$ refers to exactly one object in O.
- **Singular naming**. A naming scheme N satisfies the singular naming requirement if in each possible state σ of the world, each object in $range(N_\sigma)$ is named by exactly one proper name from V.

In other words, singular reference requires each N_σ to be a function $dom(N_\sigma) \rightarrow O$ and singular naming requires each N_σ^{-1} to be a function $range(N_\sigma) \rightarrow V$. Figure 5 shows a naming relation that satisfies these two requirements. Figure 6 illustrates violations of the two requirements. *Singular reference* is needed to exclude states of the world in which one oid refers to more than one real-world object (i.e. to exclude homonyms), and *singular naming* is needed to exclude states of the world in which one object has two different oids (i.e. to exclude synonyms).

An important consequence of the two singularity requirements imposed on object identification schemes is that if we use a single oid scheme, then in each single state of the world, we can count objects by counting their oids. This is true in *each single* state of the world, but not *across all* possible states of the world. For example, figure 7 shows two state transitions that are allowed by the singularity requirements but which we want to exclude. In order to represent historical information adequately, we must additionally impose the following two requirements:

- **Rigid reference**: After each state transition of the world, each proper name remains referring to at least the same object(s) as before.
- **Rigid naming**: After each state transition of the world, each object remains named by at least the same proper name(s) as before.

Rigid reference helps to exclude situations in which in two different states of the world, the same oid is used for different real-world objects (i.e., this excludes reuse of oids), and *rigid naming* helps to exclude situations in which in two different states of the world, different oids are used for the same real-world object (i.e., this excludes renaming).

We formulated the rigidity requirements in such a way that they are parallel to the two singularity requirements. Note, however, that they are equivalent: a violation of either one is always a violation of the other one as well. Actually, the rigidity requirements are both equivalent with the following, more precisely formulated requirement:

- **Monotonic designation**: In each pair of successive states σ_1 and σ_2 of the world, $N_{\sigma_1} \subseteq N_{\sigma_2}$.

[2] We use the term "explicitly represented" to exclude the case that an object (e.g. a potato) belongs to a quantity (e.g. 20 pounds of potatoes) registered in the administration, but is not registered individually. Such an object would be implicitly, but not explicitly represented.

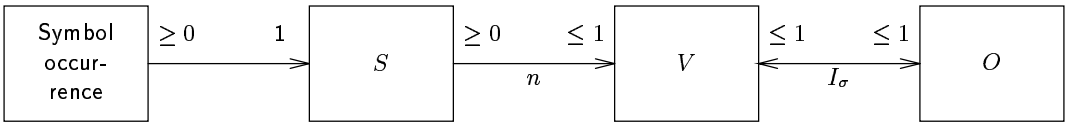

Figure 5: An oid relation I_σ satisfies the singular reference and singular naming requirements. It also satisfies the monotonic designation requirement, but this cannot be represented in this figure.

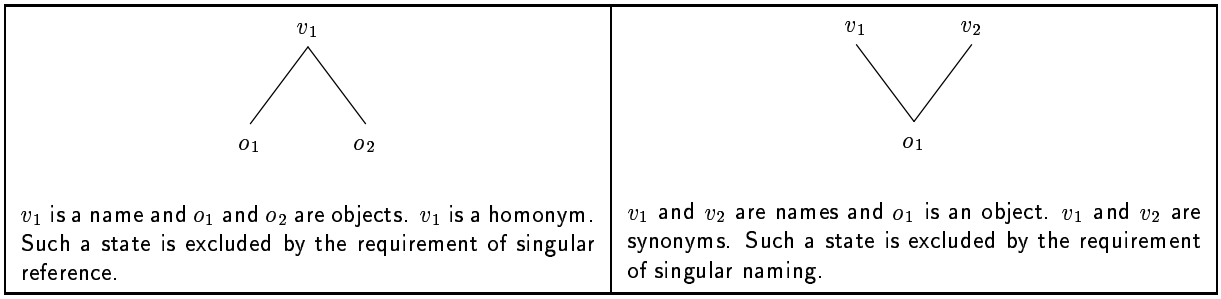

Figure 6: The situations excluded by the two singularity requirements.

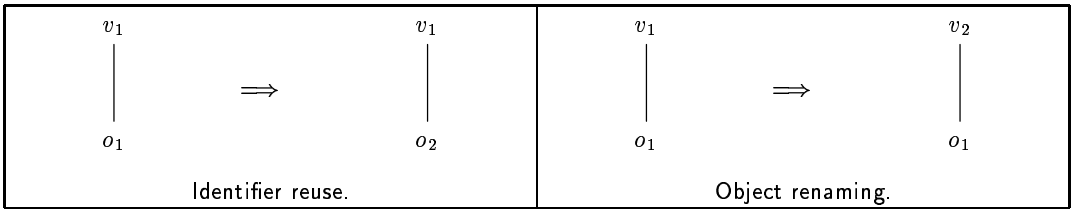

Figure 7: Two state transitions violating the monotonic designation requirement but not the singularity requirements.

Altogether, a naming scheme must satisfy three requirements to be an object identification scheme. The two singularity requirements demand that there be a 1-1 correspondence between oids and objects, and the monotonic designation requirement demands that oid assignments are never deleted.

Together, the three requirements imply that in an administration that uses a single oid scheme, we always can count objects by counting oids. In addition to allowing us to count objects in one state of the world and across historical states of the world, oids allow us to distinguish objects even if they have the same (represented) object state. We always model the world at a certain level of abstraction. This means that we may not represent enough attributes of objects to be able to guarantee that in all possible states of the world, different objects have different attribute values. In this case, we can rely on oids to tell us which objects are different.

3 Oids versus keys and surrogates

3.1 Keys

In relational databases, a **key** is a combination of one or more attributes with values that must be a unique combination in a relevant set of tuples. For example, a **database key** must be unique in each allowed state of the database, and a **relation key** must be unique in each relation instance. We call a key **user-assigned** if there is at least one database user (other than the database administrator) who may assign key values to tuples.

We have identified the following (mutually related) differences between keys and oids (see also figure 8). First of all, keys identify tuples in a database state, whereas oids identify objects in a part of the world. This means that the concept of a key is a database concept whereas the concept of an oid is more general; the definition of oids above does not mention databases at all, and therefore has wider applicability than in computerized databases alone. (It is clear though that oids are required to make good databases.)

Second, because keys consist of attributes, key values may represent information about the state of objects. For example, a key consisting of a name and birth date not only identifies a tuple uniquely in the relevant set of tuples, it also contains the name and birth date of the represented entity. In our definition of oids, we have *not* excluded the possibility that an oid represents information about the object it represents. However, because of the singularity requirements, the represented properties of each object must then be unique across all *possible* objects and because of the monotonicity requirement, they must also be *unchangeable*. (Note that it is practically impossible to find such properties. We return to this when we discuss the possibility of fraud in section 5.2.)

Third, keys are updatable whereas oids are not. It is true that updates of key values can easily be prohibited by adding to the definition of a (primary) key the requirement that key attributes must be non-updatable. However, this would merely mean that the key of a *tuple* cannot be updated; the key of the *represented entity* then still can change. If the represented entity changes a key value, we can simply delete the tuple that represents the entity and insert another one, containing the new key value.

Fourth, a key is required to be unique in each *single* state of the *database* (or relation), whereas an oid is unique across *all* possible states of the *world*. (The concept of a world is made more explicit in section 6.) This is a consequence of the monotonicity property imposed on oids (but not on keys).

A fifth difference between keys and oids often mentioned is that oids are *claimed* to avoid the information transfer problem. (As we will see, this difference is only gradual.) When an administration A receives information from an administration B, the **information transfer problem** for A is to determine on the basis of transferred information alone whether the transferred information is about an object already represented by A, and if so, which one this is. This problem arises whenever the receiver uses a different naming scheme than the sender does. This may occur in federated databases [7], in database merges [9] and in EDI networks. For example, when two vehicle databases are combined that use different keys (or oid schemes) for vehicles, it is impossible to determine on the basis of keys (or oids) alone whether two tuples represent the same vehicle.

We define the **usage scope** of a naming scheme as the set of administrations currently using the naming scheme. The usage scope of a naming scheme can be different in different states of the world. Indeed, the information transfer problem does not exist when there is one identification scheme whose object space O contains all possible objects whatsoever, and whose usage scope is always universal (i.e. that is used by all

	Keys	Oids	Internal identifiers (Surrogates)
1.	Database concept	Modeling concept	Implementation concept
2.	May carry changeable information	Carries no changeable information	Carries no changeable information
3.	Updatable	Non-updatable	Non-updatable
4.	Unique in each single state of one database (or relation)	Unique across all possible states of the world	Unique across all possible states of one database system
5.	Information transfer problem frequent	Information transfer problem infrequent	Information transfer problem between different database systems
6.	Often assigned by a database user	Assigned by an oid assignor (see section 5)	Assigned by the database system
7.	Visible to the user	Visible to the user	Invisible to the user

Figure 8: Differences between keys, oids, and internal identifiers.

possible administrations). In that case, every object about which information could ever be transferred, has an oid in this global oid scheme. Since the usage scope of this global oid scheme is universal, the sender and receiver both can use it, and so they will not have an information transfer problem. Obviously, such a global oid scheme is unattainable in practice, and even if it would be attainable, its use would be undesirable for privacy reasons. We return to this in section 6. Here, we remark that keys often have a validity restricted to one database system; other database systems often use other keys for the same objects. By contrast, oid schemes (e.g. social security numbers) typically have a large usage scope and object space. Different databases within the usage scope of such a large scale oid scheme will not experience the information transfer problem when they use identifiers from this scheme in their transfers. This means that by a suitable use of oid schemes, we can reduce the frequency with which the information transfer problem is encountered.

A sixth difference between oids and keys is also gradual. Keys are often assigned to objects by database users and oids are not. Due to the usefulness of having oid schemes with a large scope, oids are often assigned by authorities higher than the database user. Thus, the personnel department assigns employee numbers company-wide and the governmental tax department in The Netherlands assigns something called a *sofi* number, which is similar to a social security number, to identify all inhabitants of The Netherlands.

Finally, we think oids should be visible to the user. This is because oids are used by people to distinguish (the representations of) different real-world objects and to identify the same object in different states. Social security numbers, serial numbers and other proper names used as oids, would be of no use if they were invisible to people. Thus, there is no difference in visibility between keys and oids.

3.2 Surrogates and internal identifiers

Besides the concepts of oid and key, the concept of a surrogate has been proposed [4, 6]. There are really two concepts here, that of surrogate and that of internal identifier, which must be distinguished (figure 9). A real-world object o can be represented in a database system by an internal object s. We will call s a **surrogate** for o. Each state of o is represented by a state of s, which can be stored as, for example, a tuple. In particular, we can store an external identifier of o, such as a serial number nr, in the tuple.

In addition to identifying o, we also need to identify s so that it can be recognized as the same through all its possible states, and can be distinguished from any other possible surrogate in the database. This is done by an **internal identification scheme** in which **internal identifiers** i are generated by the database system and assigned as oids to surrogates. The object space of this identification scheme is the set of all possible surrogates. Internal identifiers are visible to the DBMS software, but not to the application soft-

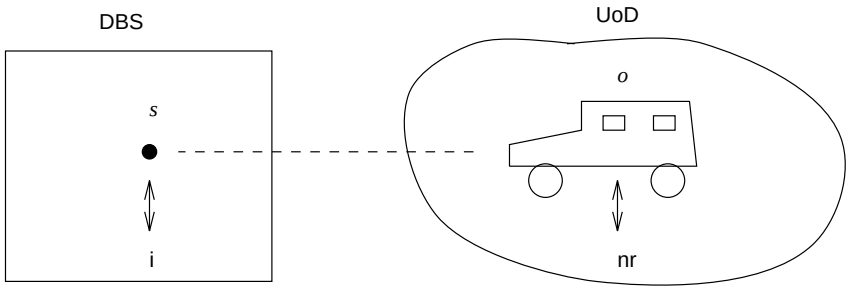

Figure 9: A database system (DBS) represents part of the world (the Universe of Discourse or UoD). The external object o is represented by an internal object s (e.g. a record), which is a surrogate for the external object o. nr is an external identifier of o and i is an internal identifier of s. Because of the 1-1 correspondence between i and s, it does no harm to call the internal identifier i a surrogate.

ware, nor to the users of the database system, nor to other systems with which it could communicate through an EDI network. Thus, internal identifiers are invisible to the user. When information about a new real-world object o is entered in the database system, a surrogate s is created and a fresh internal identifier i is generated. An example of a tuple that represents the car in figure 9 is

$$\langle int\_oid : i, ext\_oid : nr, weight : 5000, \ldots \rangle.$$

Each state change of o is represented by a state change of s, which is a replacement of the above tuple by another one. Because i and nr are identifiers, they will remain invariant through all state changes. However, note that we can replace the internal identification scheme by another one without the user of the database system noticing it. Such a replacement is not a state change of the implementation, but a replacement of one implementation by another one.

Because of the 1-1 correspondence between internal identifiers and surrogates, it can do no harm to treat internal identifiers as if they were surrogates.

Internal identifiers differ from keys, because they are assigned by the database system and not by the user. In addition, they are kept invisible from the user. Internal identifiers are implementation concepts, not database concepts. Because internal identifiers are a particular kind of oids, they carry no changeable information about the surrogates they identify, and they are non-updatable. In addition, unlike keys, they are unique across all database states. Finally, there is no information transfer problem as long as the transfer remains within the same database system. For example, transfering information between two different views of one database system is possible.

Because internal identifiers have a usage scope limited to one DBMS, they cannot be used instead of oids. Nevertheless, the concept can be useful for constructing efficient DBMS implementations.

Because of the limited usage scope of internal identifiers, the information transfer problem is even more acute for internal identifiers than it is for keys. Note that even if database administrators would enforce internal identifiers in two database systems to satisfy the singularity and monotonicity requirements across those database systems, the information transfer problem would still exist for database users, for internal identifiers are invisible to the users.

4 Some (non-)examples of oids

For concreteness, we give some examples and non-examples of oid schemes.

- *Passport numbers* do not qualify as oids for persons, for one person may have two passports (e.g. because he or she has two nationalities), which violates singular naming. In addition, some people may loose a nationality, and thus a passport, which violates monotonic designation. The oid requirements are also violated when people renew their passport and receive one with a number different from the old one.

- *Employee numbers* can qualify as oids for the object space of employees employed by one company. This works if we let employee numbers only identify employee *roles* played by people. (We will say more about roles in section 6.3.)

If an employee number would also be intended to identify the *person* playing the employee role, then we would have to enforce that nobody can have two jobs with two different employee numbers at the same company; furthermore, we would have to ensure that whenever the same person is re-employed by the same company, the same employee number is given to that person, even if this occurs many years later.

- The *Vrije Universiteit* (VU) library assigns bar codes to all its documents and all library users. According to the description of its administrative library procedures, the first digit of this bar code is a "1" if the identified object is a document and it is a "2" if the identified object is a user. The bar code has a fixed, finite length. If we assume that the library will never possess more documents than can be named by different fixed-length bar codes starting with a "1", then these bar codes can be used as an oid for documents. On the other hand, a bar code starting with a "2" is not an oid for persons, because if a person stops being a user of the library and later becomes a user again, he or she receives a different bar code. It is not even an oid for users, where $USER$ is a role of $PERSON$s, because a user can loose his or her plastic card on which the bar code is printed, and then can receive a new card, containing a different code while still being regarded as the same user. Thus, contrary to what is stated in the description of VU library regulations, the bar code is an oid of user identification cards, and not of persons or users.

- Rumbaugh [14] gives the example of a credit card company which identifies credit cards by the account number of the credit card. This gives a problem if two people (e.g. husband and wife) each have a credit card for the same account, and the credit card is used by a third party (e.g. a car rental company) to identify a person hiring a car. This is an increase of scope of this oid scheme, which causes problems if not all administrations using the oid scheme agree about what the object space of the scheme is. The rental company in this case did not realize that the object space of this oid scheme is the set of all possible credit card account numbers, and not the set of all possible credit card users. The credit card company itself did not realize this well enough either, for Rumbaugh's example shows that this company wishes to distinguish transactions performed with the two cards; this is impossible if the object space is the set of all possible account numbers rather than the set of all possible cards itself.

- *Unix process numbers* do not qualify as oids, because they violate the monotonic designation requirement. There is a finite set of available process numbers and Unix reuses numbers once all numbers have been used once. In addition, after each bootstrap, Unix restarts assigning numbers to processes starting from 1.

- *Ethernet addresses* do not qualify as oids, even though they were intended to be oids. There is a world-wide distributor that allocates finite sets of unused Ethernet addresses to different manufacturers. Each manufacturer can use numbers from its set to identify the boards it produces. If a manufacturer runs out of numbers, it can get a new set of unused numbers. However, there is no agreement among manufacturers about what the object space of this oid scheme is. Some manufacturers (call them type 1) think the object space is the set of all possible Ethernet boards and others (call them type 2) think that it is the set of all possible machines in which Ethernet boards are placed. Since one machine can have many Ethernet boards, a machine produced by a manufacturer of type 1 can have many Ethernet addresses; but different Ethernet boards in a machine produced by a manufacturer of type 2 share the same Ethernet address.

- An *internet domain name* is a proper name used to identify clusters of nodes in the Internet network. Its composition is not fixed, but in many cases it consists of a name for a country, a local network within the country and a host in this local network. This means that it holds state information. This is not strictly forbidden for oids, but we saw that this state information must then be unique and unchanging. The state represented by an internet address, however, can change: the union of Germany and the division of Czecho-Slovakia caused some domain names to change. Internet domain names are therefore not oids.

We now discuss a few general mechanisms for oid assignment, and look at a few problems with oid assignment.

5 Assignment of oids

Real-world objects are not "born" named. If we want objects to have a name, we must give it to them. To implement a naming scheme $N : \Sigma \rightarrow \wp(V \times O)$, we need an **assignor** that assigns names from V to objects from O. The simplest way to implement an oid scheme is to choose the natural numbers as value space, start assigning with 0 and let the oid assignor remember the last assigned value. Each time an object must be named, the assignor uses the next higher natural number.

5.1 Information content of oids

It is permissible to use oids with information content, as long as this content never changes. One way in which it may be useful to allow oids with information content is a situation in which we want to have more than one assignor per oid scheme. Let A be the set of assignors of a naming scheme. We must then ensure that each $a \in A$ generates a name that is not yet generated by any assignor for the oid scheme. One way to do this is to define a mapping that assigns to each $a \in A$ a set V_a of potential names, such that $V_a \cap V_{a'} = \emptyset$ for $a \neq a'$. A simple way to define this mapping is to first implement a higher level oid scheme with A as its object space. Each assignor $a \in A$ can then generate a value independently from any other assignor, but prefix this value with its own oid. Note that the values in V now have an internal structure. This can do no harm as long as we use one oid scheme for the assignors. The generated names now contain information about who generated them.

Under certain conditions it is also permissible to put class information in an oid. If C is an object class, we call C a **natural kind** if instances of C will be instances of C for as long as they exist. For example, an instance of CAR will remain a CAR instance for as long as it exists. By contrast, an instance of class $STUDENT$ is really an instance of $PERSON$ in the state of being a $STUDENT$. (Deciding whether or not a class is a natural kind is a modeling decision that may depend upon the application. For example, in some applications we may want to represent the possibility that a CAR is rebuilt into a $TRUCK$. In a model of this application, CAR is not a natural kind.)

Suppose NK is a set of natural kinds that partitions O, i.e. each $o \in O$ is an element of exactly one $k \in NK$. We then first implement an oid scheme with NK as object space and implement one assignor for each $k \in NK$. Now assignors can again operate independently from each other, but prefix the generated names with the name of the natural kind for which they generate names. For example, the object space of the VU library naming scheme has two natural kinds, $DOCUMENT$ and $USER$. The oid of documents has the structure "$DOCUMENT.n$" and the "oid" of users (which we saw is really an oid of user identification cards) has the form "$USER.n$". If there is more than one assignor for a natural kind, then we can again use the technique mentioned earlier and further prefix the generated value with the oid of the assignor itself to get n.

Prefixing with a natural kind name is allowed because an oid composed in this way does not give changeable information about the object. If, on the other hand, migration into and out of a class C would be possible, membership of C would be part of the changeable state of an object and we cannot put the name of C into oids of instances of C. For example, since we know that people will never turn into documents and vice versa, we may encode the class $DOCUMENT$ or $PERSON$ into the oids used by the library. This is even so if we view bar codes starting with a "2" more correctly as identifiers of user identification cards, for presumably, user identification cards will never turn into library documents and vice versa.

5.2 The oid presence and authentication problems

Oid assignment has some persistent problems, which we call the oid presence and authentication problems. They arise from the fact that oid assignors must be able to distinguish which objects have already received an oid, and which objects have not. This makes two kinds of mistakes possible.

First, an assignor must be able to recognize whether or not an object has an oid, i.e. is already in $range(N_\sigma)$. This is the **oid presence problem**. This problem is most severe for an oid assignor, but it also exists for users of oids. If an oid assignor can make a mistake in recognizing the presence or absence of an oid for a real-world object, then an object may receive more than one "oid" and this would violate the singular naming requirement of oids.

For users of oids the oid presence problem also exists, although it is less severe because users may know which types of objects are supposed to have been as-

signed an oid by an oid assignor by the time they encounter them. For example, we usually know that an engine block should have been assigned a serial number as oid. If we do not see a serial number in its proper place, we know that something is wrong: perhaps the assignor forgot to assign it or someone erased it.

The second problem is that the baptizing procedure in which a real-world object receives an oid can be faked. Thus, objects that have or have not received an oid from an oid assignor can receive a fake oid from a fake oid assignor. The value used as fake oid may or may not have already been assigned by an assignor as oid to an object. Fake baptizing procedures may lead to one proper name being used as "oid" for different objects or different proper names being used as "oids" for one object, thus violating the oid requirements. For example, engine block numbers can be altered and people can give a false social security number. In practice, people have been falsely accused after others used their social security number as a fake identifier. We call the problem of recognizing whether a proper name is a true or a false oid the **oid authentication problem**.

The oid presence and authentication problems can be partly prevented, but not completely solved. The only way to solve the oid presence problem is to eliminate the oid assignment process by finding a combination of properties that all objects have and that can act as oid. The desired combination of properties should therefore be singular (there should be a 1-1 correspondence between the properties and the objects) and monotonic (the properties should never change). The trouble, however, is that it is very hard or even impossible to find such a combination of properties.

For example, it is not even completely certain whether fingerprints are unique for all possible persons. But if we assume that a singular combination of properties could be found, then we would only have solved the oid presence problem, since every object now has an oid. The authentication problem then still remains, since it is impossible to prevent fraud. For example, even fingerprints might be changed, e.g. by burning or skin transplantation. Furthermore, they are not unique for all possible persons and rubber gloves taken together. It is possible to give a rubber glove the same fingerprint as a person and use this glove to mislead an identification device based on fingerprints. As far as we know, for any identification scheme, fraud is always possible (although it may be very expensive).

The oid presence and authentication problems exist in all implementations of oid schemes. They are not alleviated, and often aggravated, when we choose a hierarchical oid distribution scheme in which one central oid distributor delegates the authority to assign oids to subdistributors. This is because all oid (sub)distributors must now be able to recognize the presence of an oid assigned by *any* of them, and must be able to authenticate an oid string generated by any of them as being a legal oid.

5.3 Borrowing oids

One way to reduce the oid presence and authentication problems is to reduce the number of oid schemes that are used. We simply leave the generation and assignment of oids to one higher level institution, with a large scope, and use the oids distributed by this institution in different local administrations. For example, in The Netherlands, something called a *sofi* number is used to identify all people who are registered as inhabitants of The Netherlands. In addition, a person who is not registered as an inhabitant of The Netherlands can request a *sofi* number and, under certain conditions, be granted one by the Dutch government. Within its object space, and assuming that the Dutch government really succeeds in preventing any person in the object space to obtain more than one *sofi* number, the *sofi* number satisfies all oid requirements. Because the *sofi* scheme has a large usage scope (many administrations in The Netherlands use it) and it has a reasonably large object space, it is a suitable candidate from which to borrow oids.

Borrowing oids reduces the oid presence problem, because users of oids often know that certain kinds of objects are supposed to have an oid. It does not reduce the oid authentication problem. In addition, borrowing oids does create a dependence on some high level institution. For example, some companies in The Netherlands have foreign nationals as customers. These have not always received a *sofi* number, so these companies cannot use *sofi* numbers as oids for their customers. In general, only objects that have been assigned an oid by the higher level institution can be represented, and it is often not possible for a company to influence the naming policy of this higher-level organization to suit the needs of the company exactly.

6 Information transfer

We have seen several times that there is not one oid scheme for all possible objects in the world, but that there are many different oid schemes, often with different but overlapping object spaces, value spaces, usage scopes, and notation systems. This creates a problem with information transfer. The information transfer problem has been defined above as the problem for an administration A, when it receives information from an administration B, to determine on the basis of transferred information alone whether the transferred information is about an object already represented by A, and if so, which one this is. To explain the problem in more detail, we must make the concept of *world*, used in table 8, more precise. We will do this by replacing it with the more technical term "Universe of Discourse". A more accurate reading of table 8 ensues if we replace "world" with "universe of discourse" as defined below.

6.1 The UoD of an administration

The **Universe of Discourse** (UoD) of an administration is the set of possible objects that are of interest to the administration and about which the administration may represent information. If we connect two administrations by merging them or by connecting them through an EDI network, then we do this with the intention to join at least part of their UoD's. If the UoD's are known to be disjoint, then there is no information transfer problem, for the answer to the question whether the received information is about an object already in A, is always negative. If the UoD's are disjoint, then merging two administrations gives no problem even if the oid schemes use the same value space, since we required oids to be disambiguated by subscripting them with the naming scheme in any case.

If, on the other hand, the joined UoD's overlap and the two administrations are not known to use the same oid scheme(s) on the objects in the intersection of the UoD's, then there is an information transfer problem. Given two oids from different oid schemes, it is impossible to ascertain on the basis of the oids alone, whether they name the same object. In general, looking at the attributes stored with the oids does not give a definite answer to the identity question either; we can achieve certainty only by looking at the corresponding object(s) themselves.

The only way to avoid (current or future) information transfer problems is to use oid borrowing as much as possible, so that as many different administrations as possible use the same oid scheme for the same object space. This explains the tendency for the usage scope of oid schemes to grow, as well as the tendency to define oid schemes for object spaces that are as large as possible. For example, if two companies merge their employee administrations, then they can do this effectively if they both used the *sofi* number as employee identifier. However, if the identified objects are people, the use of a large object space and a large usage scope goes counter to the demand for privacy of human beings [3, 13].

6.2 Privacy

There are several forms of privacy. For example, bodily privacy puts limits on the situations and ways in which people can be physically examined, and spatial privacy puts limits on the situations in which people's homes can be searched. We are concerned here with **information privacy**, which is defined by Westin [16] as "the claim of individuals, groups or institutions to determine for themselves when, how and to what extent information about them is communicated to others." We adopt this definition here.

6.3 Protecting privacy by introducing roles

Administrations have a tendency to increase the possibility of information transfer by enlargening the usage scope of an identification scheme, so that more administrations use common identifiers. This tendency should be balanced against the claim of people and institutions to information privacy. A simple way to combine the need for identification with the need for privacy is to define **roles** of objects and identify the roles instead of the objects that play them. We argue elsewhere that roles are needed for another purpose too, viz. to solve an apparent paradox of classification [17, 18, 19]. In the following paragraphs, we very briefly describe what this paradox is.

A class like $EMPLOYEE$ is often said to be a subclass of the class $PERSON$. If so, each $EMPLOYEE$ would be identical to a $PERSON$. However, there are people who have two or more jobs. In this situation, we would like to be able to say that e_1 and e_2 are two *different* employees, with different employers, salaries, employee numbers etc., but that they are the *same* person. But if this is so,

EMPLOYEE cannot be a subclass of *PERSON*, because this implies that each *EMPLOYEE* is identical to a *PERSON*. And since the identity relationship is transitive, if e_1 and e_2 are both identical to the same person, they would be identical to each other.

The paradox can easily be resolved by modeling *EMPLOYEE* not as a subclass of *PERSON*, but as a **role class** of *PERSON*. The instances of *EMPLOYEE* are roles **played by** *PERSON* instances. Each role has its own identifier, and we can simply associate a role identifier to an object if the object plays that role. This is not the whole story, for we may want to take care of (some kind of) inheritance. The application of a *PERSON* attribute (e.g. name or birth date) to an *EMPLOYEE* role should perhaps not give a type error, but must return the value that the attribute has for the person who plays the employee role. As is discussed in more detail elsewhere [18], this could be accomplished by a delegation mechanism [12, 15].

Returning to the subject of privacy, if the requirement of anonymity of persons must be combined with the need for identification, then this is done by identifying *roles* of persons and not the persons themselves. Thus, passengers get a ticket which may identify them as a passenger, but not as a person.

The use of roles played by people to represent information about them allows us to keep the people who play these roles (partly) anonymous. The connection between roles and the people who play them could for example be stored in a secure place. This could be realized by "privacy banks", to whom we would give the connection between our oid and our role identifiers. Just as we usually spread our money over a few banks, we would spread the information about our role identifiers over a few different privacy banks. Privacy banks should be subject to strict rules and regulations about the way they should guard these secrets. The connection between a person and the roles he or she plays should only be given away under strict conditions, because it is a handle to dossier linking [2]. Thus, the usage of oids identifying roles reduces (as it is intended to do) the possibility of transferring information about these people or institutions.

7 Comparison with other work

The concept of surrogate has been defined in 1976 (roughly in the sense of internal identifier) by Hall, Owlett and Todd [6, pages 206-207] as a solution to the following problems:

- Updating keys without losing track of the identity of the represented object.
- Distinguishing real-world objects that happen to be represented in the system by the same attribute values.
- Representing information about an object before it has been assigned a key value.

Hall et al. attribute this idea to Engles [5]. Another early appearance of the same idea is Cadiou [1].

Internal identifiers are used by Codd in RM/T [4] (where they are also called surrogates). In addition to the arguments above, Codd adds a fourth one:

- It is possible that one entity is represented in two (relational) tables with different user-assigned keys [4, pages 409–410]. Without internal identifiers, identity information would then not be represented, not even in one database system.

Khoshafian and Copeland [9, page 408] use the same arguments as above to argue for the need for object identifiers. In addition, they use an argument based on information transfer:

- If companies merge their databases systems, which use different oid schemes, then the oid scheme of one of the database systems must change. But this would cause a discontinuity in the identity of the objects represented by that database system.

A solution to this problem mentioned by Khoshafian and Copeland is that both database systems use internal identifiers and continue to use these internal identifiers after they are merged. This avoids the discontinuity problem, but does not solve the information transfer problem. A better solution would be to prevent the information transfer problem (and therefore the discontinuity problem) by letting both database systems use from the beginning the same (visible) oid scheme, whose object space is large enough to include all objects in the intersection of the UoD's of both database systems. As pointed out earlier, the desirability of this solution must be balanced against privacy claims.

After Khoshafian and Copeland's paper, the term "object identifier" came to be used for a concept that, for different authors, has different degrees of overlap

with our concept of internal identifier. This is regrettable, because a lack of a consistent terminology leads to a confusion of distinct concepts, which leads to a lack of recognition of the requirements for good naming schemes, and therefore results in bad naming schemes.

Kim [10, page 108–109] uses "object identifier" more in the sense of "internal identifier". For example, his "identifiers" are system-generated and invisible. They also contain the name of the class of the identified object, which we saw is permissible if objects cannot migrate between classes. However, in Kim's approach, objects are allowed to migrate to a different class, and if migration occurs, the class name in the "identifier" is replaced by the name of the class to which the object has moved. Since his class membership is changeable, the name of the class cannot be stored in any oid. In particular, it cannot be stored in an internal identifier.

Kim [11, page 681] correctly observes that the use of (invisible) internal identifiers does not eliminate the need for keys. However, we hope to have shown that keys do not appropriately solve the problems of object identification, and that the use of (visible) oids does make it possible to dispense with keys, in other words, that object identity should be represented by oids.

Kent [7, 8] uses the term "identifier" almost in the way defined in this paper. He mentions the following four identifier criteria [7, page 12]:

1. Identifiers should be unique (this is our singular naming requirement),
2. singular (this is our singular reference requirement),
3. total (each object should have one), and
4. stable (this comes close to our monotonic designation requirement).

One difference with our view of oids is only that we do not require oid schemes to be total: we also allow anonymous objects. This difference is probably due to the fact that Kent is concerned with the problem of connecting information about objects stored in different, heterogeneous databases, whereas we are also concerned with the identification of objects in a UoD outside databases. In those UoD's, there exist unidentified objects.

Another difference is that we make a distinction between symbols and values and use values as oids. Kent does not make this distinction and uses his symbols mostly as we use values [8].

8 Conclusions

In this paper, we gave a precise definition of the concept of an oid scheme as a naming scheme that satisfies the singularity and monotonicity requirements. We listed a number of differences with the concepts of key and surrogate, and defined internal identification schemes as a special kind of oid scheme. We listed a number of ways in which oid schemes can be implemented, and listed a number of practical limits to their use. The oid presence problem is the problem for an oid assignor to ascertain that an object has already been assigned an oid, and the oid authentication problem is the problem of ascertaining that a value is a (legally assigned) oid. We noted that the information transfer problem is solved within one object space of one oid scheme, but not across object spaces of different oid schemes. The frequency with which the information transfer problem is encountered can be reduced if we make object spaces and usage scopes of oid schemes as large as possible. However, the tendency of administrations to increase the usage scope of oids should be balanced against the claim of people and institutions to information privacy. One way to do this is to use role identifiers, which allows one to keep the disclosure of the connection between a role and a person under one's own control.

Acknowledgements: We thank Hans van Staveren for helping us with the intricacies of the naming schemes of Internet, Ethernet boards and Unix processes. Thanks are due to Henri Bal and Andy Tanenbaum for comments given on a draft version of this paper. The detailed comments of the anonymous referees helped us to rethink some fundamental points, which led to improvements in the presentation of the paper.

References

[1] J.M. Cadiou. On semantic issues in the relational model of data. In *Proc. 5th Symposium on Mathematical Foundations of Computer Science*, pages 23–38. Springer, 1976. Lecture Notes in Computer Science 45.

[2] D. Chaum. Security without identification: transaction systems to make Big Brother obsolete. *Communications of the ACM*, 28:1030–1044, 1985.

[3] R.A. Clarke. Information technology and dataveillance. *Communications of the ACM*, 31(5):498–512, May 1988.

[4] E.F. Codd. Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems*, 4:397–434, 1979.

[5] R.W. Engles. A tutorial on database organization. In *Annual review of Automatic Programming*, pages 1–64. Pergamon, 1972. Volume 7, part 1.

[6] P. Hall, J. Owlett, and S. Todd. Relations and entities. In G.M. Nijssen, editor, *Modelling in Database Management Systems*, pages 201–220. North-Holland, 1976.

[7] W. Kent. The breakdown of the information model in MDBs. *Sigmod record*, 20(4):10–15, December 1991.

[8] W. Kent. A rigorous model of object reference, identity, and existence. *Journal of Object-Oriented Programming*, 4(3):28–36, June 1991.

[9] S.N. Khoshafian and G.P. Copeland. Object identity. In *Object-Oriented Programming Systems, Languages and Applications*, pages 406–416, 1986. SIGPLAN Notices 22 (12).

[10] W. Kim. *Introduction to Object-Oriented Databases*. MIT Press, 1990.

[11] W. Kim. Object-oriented database systems, promises, reality, future. In *Proceedings of the 19th International Conference on Very Large Databases*, pages 676–687, 1993.

[12] H. Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In N. Meyrowitz, editor, *Object-Oriented Programming: Systems, Languages and Applications*, pages 214–223, October 1986.

[13] M. Rotenberg. Prepared testimony and statement of Marc Rotenberg, Director, Washington Office, Computer Professionals for Social responsibility (CPSR), on the use of the social security number as a national identifier, before The Subcommittee on Social Security, Committee on Ways and Means, U.S. House of Representatives, February 27, 1991. *Computers and Society*, 21:13–19, October 1991.

[14] J. Rumbaugh. A national identity crisis. *Journal of Object-Oriented Programming*, 5(6):11–16, October 1992.

[15] E. Sciore. Object specialization. *ACM Transactions on Information Systems*, 7(2):103–122, 1989.

[16] A.F. Westin. *Privacy and Freedom*. Atheneum, 1970.

[17] R.J. Wieringa and W. de Jonge. The identification of objects and roles. Technical Report IR-267, Faculty of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, December 1991.

[18] R.J. Wieringa, W. de Jonge, and P.A. Spruit. Roles and dynamic subclasses: a modal logic approach. In M. Tokoro and R. Pareschi, editors, *Object-Oriented Programming, 8th European Conference (ECOOP'94)*, pages 32–59. Springer, 1994. Lecture Notes in Computer Science 821.

[19] R.J. Wieringa, W. de Jonge, and P.A. Spruit. Using dynamic classes and role classes to model object migration. *Theory and Practice of Object Systems*, 1, to be published. Extended version of [18].

Contents