JOURNAL OF OBJECT TECHNOLOGY

Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2002**

Vol. 1, No. 3
Special issue: TOOLS USA 2002 proceedings

Object Oriented Extension to Time Series Model

Dilip Patel, Shushma Patel, and Paul Schleifer, South Bank University, UK

Abstract

It has been widely observed that temporal semantics and functionality are often developed on an *ad hoc* basis, and the benefits of temporal databases research are rarely realised. In this paper we propose an independent temporal model, which embraces object oriented concepts and also show how UML can be used to model temporal business concepts.

1 INTRODUCTION

Computer hardware technology has evolved such that manipulating large data sets is no longer problematic. Furthermore, the remaining problems introduced by the use of relational databases can be ameliorated by the adoption of an object-oriented technology, which also facilitates re-use. There are two reasons why temporal databases research might not feature in business applications that require temporal semantics and these are:

- No consensus temporal model has been accepted by the research community [Pissinou 1993]. In this paper we propose an independent model that can then be interpreted in terms of the core modelling features available under the object-oriented database model and satisfy six of the eight temporal principles proposed by Pissinou and Makki [1993].
- The absence of modelling tools and notations. We adopt a formal specification language, adapted from VDM [Jones 1986] and UML to accommodate the modelling of object-oriented concepts.

2 ADAPTATION OF THE TS MODEL TO SATISFY TEMPORAL PRINCIPLES

The TS model [Segev 1987] is a physically independent temporal data model that satisfies many of the Temporal Principles proposed by Pissinou and Makki [1993]. The Temporal Extension Principle is not applicable to the TS model because it is not an extension of any underlying non-temporal model. The aspects of the Temporal Principles that are not satisfied by the TS model are addressed in this section: transaction-time-stamping and schema evolution (Temporal Evolution Principle); branching valid time (Temporal Representation Principle); relative valid time (Temporal Incompleteness Principle).

Valid-Time and Transaction-Time-Stamping in the Adapted TS Mode

The TS model considers temporal data as a three-dimensional object which can be represented as a <*S, (T, V)**> time sequence collection (TSC). In this representation, only one dimension of time is provided, and this dimension of time is usually considered to be that of valid-time. This means that the state of an entity cannot be associated with both a valid-time-stamp and a transaction-time-stamp, both of which are necessary if the resultant model is to be considered truly temporal.

The <*S, (T, V)**> notation is rejected in favour of a formal specification language, adapted from VDM [Jones 1986] to accommodate object-oriented concepts. This VDM-like language is used to develop a formalised model of how objects may be time-stamped in both the valid-time and transaction-time dimensions in a temporal database. Formal specification languages, like VDM, can be used to create precise, unambiguous descriptions for the behaviour of software. These formal specifications are abstract in that they do not restrict development to any particular language or computational model. Most existing formalisms of temporal database behaviour have been created using the notation of relational theory, and so the specifications described in this section are more abstract and independent of underlying data models than those provided by other researchers.

Basic Domains

Time can be modelled as a countably infinite, ordered set of discrete time points. The fact that time may actually be a continuous dimension is actually irrelevant since *real* numbers in a computer-based representation are only conceptually so, and their underlying representation is actually discrete. Let the domain ***T*** of time be:

$$T = \{t_{-\infty}, .., t_{now}, t_{now+1}, \ldots, t_{+\infty}\}$$

The value of $\boldsymbol{t_{now}}$ is dependent on the current time in the modelled reality and effectively divides the domain of time into *past* and *future* sets of values. The domains of valid time, ***vT***, and transaction time, ***tT***, may be defined as:

$$vT \subseteq T$$

$$tT \subseteq T$$

A function can be defined to map a time onto the set of natural numbers (***Z***), which can be realised as a simple map indexing function whose signature is given by:

$$MapToInteger{:}\, T \times G_t \to Z$$

where $\boldsymbol{G_t}$ is an argument to represent the time granularity (it should be noted that this signature represents a pair of arguments rather than a product). This argument is omitted in subsequent definitions for clarity, but it should be noted that a temporal object must be associated with a granularity argument if its time-stamp is to be mapped to the set of natural numbers.

Let ***O*** to be the set of all objects, whether composite, atomic, or collection (*i.e*., Bag, Set, Array, *etc*.). It should be noted that this is an all-encompassing, *recursive* definition in that a composite object, which includes an attribute which is constrained to the ***O*** domain, is also itself a member of the set of ***O***. This kind of definition is not strictly allowable in VDM, but is appropriate for the purposes of this discussion.

Although a temporal object our research is considered to have more semantics than a time-stamp. A generic, composite, time-stamped object type can be specified as:

TimeStampedObject ::object : **O**

time : T

Composite objects in VDM are associated by default with an appropriate make-function which can be used to create instances of the composite data type. The signature for a make-function for creating time-stamped objects, inserting any kind of object into the *object* field, is given by:

$$mk\text{-}TimeStampedObject{:}\, O \times T \to TimeStampedObject$$

It is clear that this make-function can be used to time-stamp any kind of object with any particular notion of time, since the first argument belongs to the domain ***O*** and the second belongs to the domain ***T***, of which valid time and transaction time are both subsets. Consider the following example in which an erroneous value for an object, “hxllo”, is corrected to a value of “hello”:

tso1 = mk-TimeStampedObject(“hxllo”, *tt1*)

tso2 = mk-TimeStampedObject (“hello”,*tt2*)

This is an example of a transaction-time-stamped object whose original value, stored in the database at transaction time *tt1*, has been corrected during a consequent database transaction at time *tt2*.

By substituting a valid time for a transaction time, it is possible to model a different kind of database update:

tso2 = *mk-TimeStampedObject* ("goodbye", *vt2*)

tso1 = *mk-TimeStampedObject* ("hello", *vt1*)

This shows a valid-time-stamped object whose value in the modelled reality at time *vt1* has changed from the value "hello" to the value "goodbye" at time *vt2*, and this change is reflected by the database update.

It is useful to note in this example that, when dealing with the valid time dimension, the order in which updates are made to the database need not reflect the order in which the changes take place in the modelled reality. Valid times are supplied by the components of the application system which monitor changes in the modelled reality, and any required ordering of objects in the valid time dimension can be imposed by a sorting algorithm.

This make-function can also be called recursively, thus providing support for multiple dimensions of time, though this is a deviation from the VDM specification language. For example:

2*tso* = *mk-TimeStampedObject* (*mk-TimeStampedObject* (`myObject`, *tt*), *vt*)

In this example, `myObject` is associated with both a valid-time-stamp and a transaction-time-stamp, and so *2tso* is a true ***temporal object***. The fields can be retrieved with appropriate selectors or projection functions. For example:

time (2tso) = vt [retrieval of valid-time-stamp]

time (object (2tso)) = tt [retrieval of transaction-time-stamp]

object (object (2tso)) = *`myObject`* [retrieval of original object]

Time-Stamping Complex Objects

Consider a composite object, which is used to model a *Person* entity in a company database:

Person ::	*name*	: `char*`
	house	: $\mathbf{N}_1$
	street	: `char*`
	city	: `char*`
	salary	: $\mathbf{R}$
	job	: `char*`
	manager	: *Person*

If there were a requirement to make instances of the *Person* entity temporal, the most straightforward approach might be to time-stamp the entire entity:

p = mk-Person (“Jane”, 3, “Hill St”, “Perth”, 21678.32, “Programmer”, *mp)*

tso = mk-TimeStampedObject (*p*, *vt*)

However, this (tuple time-stamping) approach may result in a great degree of data redundancy and therefore may incur unacceptable storage overheads because, in this modelled reality:

1. The *name* of a person may never change.
2. Moving to a different *house*, *street*, or *city* might not affect a person’s *salary*, *job*, or *manager*.
3. Point 2 might also be true of every field except *name*, which never changes (Point 1).

By time-stamping the whole *Person* object, we must create a complete copy of all but one field every time a single field is updated, whether in the valid time or the transaction time dimension.

An alternative (attribute time-stamping) approach is to time-stamp only the fields of an entity, which are allowed to vary with time in the modelled reality. Thus in the *Person* entity, all fields except *name* might be individually time-stamped in the valid time dimension, and so if a *Person* object is assigned a new manager, this fact can be recorded without repeating all other fields, with unchanged values, in a new, valid-time-stamped object.

However, this approach may also be unsatisfactory if, for example, a person moves to a new *house*, it is likely that the *street*, and possibly *city*, will change too. By relying on a mechanism that individually time-stamps all of the attributes of a composite object that at a particular time, there is a risk that time-stamping information will be stored redundantly, as shown below:

TShouse2 = mk-*TimeStampedObject* (*house2*, ***vt2***)
TSstreet2 = mk-TimeStampedObject (*street2,* ***vt2***)
TScity2 = mk-TimeStampedObject (*city2,* ***vt2***)

In this example, each attribute that is used to represent the person's address in the modelled reality is stamped with the same valid time, and so the same valid time must be recorded three times in the database.

A better solution to this problem is to decompose the original composite object into two new composite objects, in which each attribute is “temporally linked”. That is the value of each attribute of the entity in the modelled reality is guaranteed or, at least, is extremely likely to change at the same time. Returning to the example of a *Person* object, it is possible to split the original set of attributes into two smaller complex objects. The first is *Person2*, by which the important details of a company employee can be represented:

Person2 ::	*name*	: `char*`
	address	: *Address*
	salary	: **R**
	job	: `char*`
	manager	: *Person2*

The second complex object is that of *Address*, which can be used to capture the data pertaining to an employee's home address:

Address ::	*house*	: $\mathbf{N}_1$
	street	: `char*`
	city	: `char*`

Using this approach, a person changing their address in the modelled reality can be represented with only one valid time-stamp instead of three:

TSaddress2 = *mk-TimeStampedObject* (*address2*, ***vt2***)

A Heuristic Approach to Designing Temporal Classes Based on Valid Time Dependency

As can be seen from the *Address* example, composite objects can be designed on the basis of whether their fields are dependent on valid time. That is to say, if a set of attributes belonging a composite object are all guaranteed or *likely to* undergo value changes in the modelled reality at the same valid time as each other, then in a temporal or historical database application, then that set of fields should be grouped together in a discrete, composite object that can be valid-time-stamped independently of the other fields in the composite object.

This condition can be defined as a Boolean-valued function, ***isSynchronous***, whose signature is given by:

$$isSynchonous\text{: set of } O \rightarrow B$$

An implicit definition for this function is given by:

$$\begin{aligned} & isSynchonous(\mathrm{s})\underline{\Delta} \\ & \quad \forall t, t' \in T, \forall m, m' \in s \\ & \qquad (m \xleftarrow{f} t \neq m \xleftarrow{f} t') \Rightarrow \\ & \qquad\quad (m' \xleftarrow{f} t \neq m' \xleftarrow{f} t') \end{aligned}$$

where $m \xleftarrow{f} t$ is a mapping which gives the value of object $\boldsymbol{m}$ at time $\boldsymbol{t}$, and $\boldsymbol{s}$ is the set of objects. It is relevant to note that VDM lacks a facility for associating the value of an object representing a real-world entity with a real-world time. Temporal database researchers refer to the fact that the temporal dimension is frequently excluded in the modelling of information systems, but it is also the case that the temporal dimension is excluded even in formal specification languages.

In other words, given a *synchronous* set of objects, then for all times, if the value of one member of the set changes at any time, the value of all other members of the set will also change.

Thus, if the ***isSynchronous*** function holds in the valid time dimension for a subset of fields belonging to a composite object, then that subset of fields should be used to compose a discrete object because:

1. The storage requirements of these kinds of data in a temporal or historical database application will be reduced.
2. A semantic link probably exists between these two fields, justifying their definition as part of a discrete object.

However, there is a *caveat* that must be considered in relation to the ***isSynchronous*** function. Consider a composite object, which models a two-dimensional polygon, such as a rectangle, as four attributes of type *Point*:

Rectangle ::	*vertex1*	: *Point*
	vertex2	: *Point*
	vertex3	: *Point*
	vertex4	: *Point*

The attributes of the *Rectangle* object are conceptually part of the same object and, if the entity in the modelled reality they represent is displaced spatially, then the value of each attribute will change, thus satisfying the ***isSynchronous*** function. But if the real-world rectangle entity is rotated about its first vertex instead of being displaced, then its representation in the database will indicate that *vertex1* is unchanged and the ***isSynchronous*** function is unsatisfied.

Similarly, in the previous example in which a company employee changes their address in the real-world, it is possible that their new address will be in the same city, in a street with the same name, or possibly to a house with the same number as their previous address.

It is therefore necessary for the semantics of the real-world situation to be taken into account when complex objects are being designed using the heuristic of synchronous attributes. Although a person may be moving to an address, which shares many of the same attributes as their previous address, it is none-the-less a *new* address; a rotated rectangle occupies a *different* position even though one of its vertices has the same value as before the rotation. Conceptually, the entire complex object has changed its state following an event in the modelled reality in both examples, and although there may be a degree of redundancy in the database representation, the semantics of the design are still valid.

Sets of Objects and Transaction Time Dependency

During a single database transaction, many different database objects may be created, modified, or even deleted (deletion in a temporal database may be indicated by the addition of some kind of deletion token rather than an actual removal of data). The objects affected during a single database transaction may be considered as a set of objects whose values are all dependent on the same transaction time — the objects are *synchronous* with regard to transaction time.

In the same way that a set of fields can be associated with the same valid-time-stamp if they are synchronous with regard to valid time, so too can a set of objects modified during the same database transaction be associated with the same transaction-time-stamp. Once again, there is a potentially great economy of storage requirements to be realised. For example:

tso1 = mk-TimeStampedObject ("hello", ***tt1***)
tso2 = mk-TimeStampedObject ("goodbye", ***tt1***)
tso3 = mk-TimeStampedObject ("I'm late", ***tt1***)

can be replaced by:

tso1 = mk-TimeStampedObject ({"hello", "goodbye", "I'm late"}, ***tt1***)

In this example, only one transaction-time-stamp is required instead of three, and so in a transaction, during which many different objects are updated, there is a great potential economy of storage.

However, consider the case of the corrective update of a complex object in which only one attribute value is replaced with a corrected value:

`[initial database update transaction]`
p = mk-Person ("Jxne", 3, "Hill St", "Perth", 21678.32, "Programmer", *mp*)
tso1 = mk-TimeStampedObject (*p, tt1*)

`[subsequent database corrective update]`
p = mk-Person ("Jane", 3, "Hill St", "Perth", 21678.32, "Programmer", *mp*)
tso2 = mk-TimeStampedObject (*p, tt2*)

In this example, the *name* attribute is originally given an incorrect value ("Jxne") which is updated with the correct value ("Jane"), and the attributes representing the address, salary, job title and manager of the modelled entity are stored redundantly. However, this research takes the view that this deficiency is minor. In most applications, corrective updates are likely to be rare in comparison with the simultaneous creation and valid-time-based updates of large sets of objects, and so the storage benefits of transaction-time-stamps that are shared between database objects outweigh the overhead of the loss of transaction-granularity in corrected complex objects.

It should be noted that the objects grouped under a single transaction-time-stamp in this way are not related to each other in the modelled reality, but are semantically linked in that they have all been affected during the same database transaction. Consider a transaction in which a *Person* object stored in the database is modified in the following way:

1. The entity in the modelled reality is an employee who has moved to a new address.
2. The entity in the modelled reality is an employee who has been assigned a new manager.

This transaction can be realised in the following way:

vtStampedAddress = *mk-TimeStampedObject* (*aNewAddress*, *vt2*)
vtStampedManager = *mk-TimeStampedObject* (*aNewManager*, *vt3*)
ttStampedObjects = *mk-TimeStampedObject*
({*vtStampedAddress*, *vtStampedManager* }, *tt2*)
update-Database (*database*, *ttStampedObjects*)

Schema Evolution

The TS model does not provide a method for handling schema evolution, which is the process by which structural changes to the metadata of objects can be accommodated. Examples of this kind of structural change include the addition and removal of attributes, and changes to the domains of objects and their attributes. Database schemata are often regarded as stable, fixed metadata, but in real-world applications changes to a database schema are commonplace; errors arise in the modelling of a business enterprise, and changes may occur in the modelled reality, perhaps due to legislative reforms or to business process re-engineering. Schema evolution is the process by which such structural changes can be accommodated within a database application without a complete re-implementation or the invalidation of existing data.

Two strategies for schema evolution exist. In *class modification* [Banerjee 1987], existing class metadata are adapted to generate new definitions. Instances of the modified classes, which were created before the changes to the schema, are migrated to match the new class definition, thus ensuring backward compatibility. However, client applications designed to utilise a particular schema version may require modification following this kind of schema evolution, and multiple versions of the schema cannot simultaneously co-exist. This implies that proactive and alternative schemata cannot be used by a temporal database application that supports only this model of schema evolution. Class modification is the model of schema evolution supported by GemStone, the platform used in this research.

In *class versioning* [Skarra 1986], the metadata defining the modified class before schema evolution is preserved and so multiple class definitions may co-exist. This ensures both the backward compatibility supported by the class modification model and forward compatibility of client applications designed to access data created under a superseded schema. Furthermore, a schema version can be designed and implemented proactively, such that anticipated changes in the modelled reality can be built into a database application before they are effective, which greatly extends the capacity of a temporal database to capture speculative and predictive data.

Metadata can be modelled as database objects; for example, class definitions in Smalltalk/DB are modelled as instances of the "Class" object class. These objects can be time-stamped in the valid-time and transaction-time dimensions in the same way as any other kind of object. This is necessary because database schemata evolve in the transaction-time dimension due to changes in how the data are modelled, and schemata evolve in the valid-time dimension due to changes in the real world. The challenge of developing a mechanism for schema evolution is therefore the problem of how to map an object to the correct metadata.

Branching Valid Time

The TS model does not address the semantics of branching time in the valid-time dimension. The semantics of branching valid time can be captured by the representation of more than one time sequence associated with the same real-world entity. In an object-oriented model, this can be realised by storing all the time sequences representing the alternative states of a particular real-world entity in a containing collection. Each alternative time sequence is associated with the same object identifier, which is the identifier of the containing collection object.

Relative Valid Time

In the <*S, (T, V)**> notation used by the TS model, no distinction is made in the time dimension *T* with regard to absolute time and relative time. In some applications, only the order of observed states of a real-world entity need be preserved; absolute values of time may be considered irrelevant or may be unknown to the observer.

3 MODELLING TEMPORAL OBJECTS

Using standard UML notation, it is possible to represent an object schema which associates an instance of an Employee object class with an ordered history of snapshot

Address objects, as shown in Figure 1. However, there are no specifications for temporal semantics like valid time, transaction time, or interpolation functions.

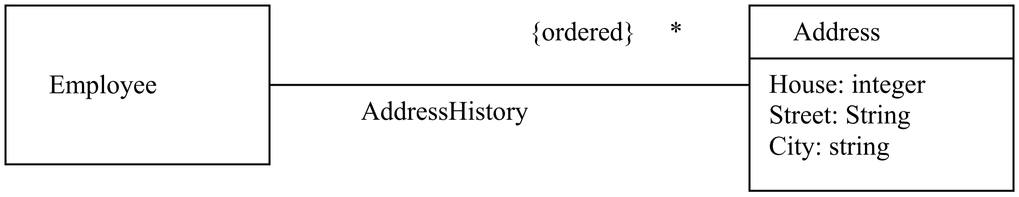

Fig 1: Employee Class with address-history Attribute

A notational solution to this problem is to use a specialised temporal link class to define link attributes as shown in Figure 2. By specifying the Address class as a snapshot, this approach shows that Address instances are snapshot objects stored in a temporal object, the semantics of which are specified by explicitly valued link attributes.

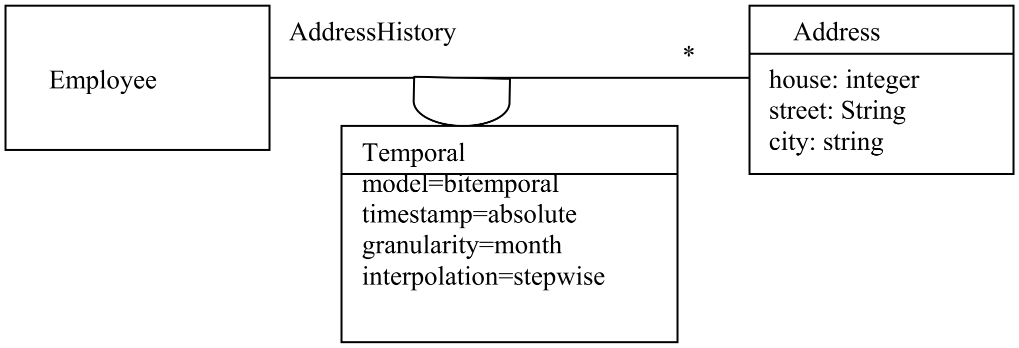

Fig 2: Employee Class with Temporal Link Attributes

The domains of the temporal link attributes are shown in Table 1. The model used for the temporal object depends on the functionality required by the application; some data may be temporally static, but it may be necessary to preserve a history of updated errors. In other applications, only a record of how the data vary in the modelled reality may be required. The domain of the model attributes has a cardinality of three to accommodate these notions.

Temporal Attribute	**Domain**
model	{rollback, historic, temporal}
timestamp	{absolute, relative}
granularity	{second, day, month, …}
interpolation	{stepwise, …..}

Table 1: Temporal Link Attribute Domain

The timestamp attribute reflects whether the application requires valid times to be stored as absolute time-stamps or if only the relative order of changes in the modelled reality is required. The granularity and interpolation attributes domains are not explicitly defined because these are extensible.

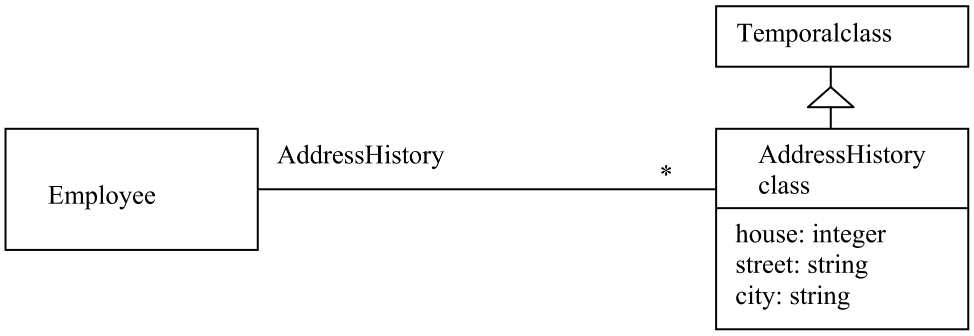

Fig 3: Temporal Class Modelling Using Inheritance

An alternative way to model temporal object schemata is to include the temporal class as a super class, as shown in Figure 3. This solution is less satisfactory because it implies that an application must use class inheritance to confer temporal semantics. This may not be possible for some object schemata implemented in an object-oriented database that does not support multiple inheritance, and in some application a compositional approach to creating complex objects might prove more effective.

4 CONCLUSION

In our research we have adopted a specific temporal model for its clear abstraction of temporal semantics and extended to satisfy the framework of temporal principles[Schleifer 1997]. The extended temporal model is described in a formal specification language. We have successfully implemented the model within an object-oriented database. The implementation is designed to minimise the storage of redundant information and encapsulates a rich set of temporal semantics that is opaque to an application programmer.

REFERENCES

[Banerjee87] Banerjee, J., and Kim, W.: Semantics and implementation of schema evolution in object-oriented databases. *Proceedings of the ACM SIGMOD Conference*, San Francisco, USA, pp.311-322. 1987.

[Jones86] Jones, C.B.: *Systematic Software Development Using VDM*. Prentice Hall. 1986.

[Pissinou93] Pissinou, N., and Makki, K.: Separating semantics from representation in a temporal object database domain. *Proceedings of the Second International Conference on Information and Knowledge Management*, Washington, DC, USA, pp.295-304. 1993.

[Schleifer97] Schleifer, P.: A Chronicle Approach to Modelling Temporal Database Objects. PhD thesis awarded at South Bank University, UK, April 1997.

[Segev87] Segev, A., and Shoshani, A.: Logical modelling of temporal data. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Francisco, CA., pp.454-466. 1987.

[Skarra86] Skarra, A.H., and Zdonik, S.B.: The management of changing types in an object-oriented database. *ACM SIGPLAN Notices* **21**(11):483-495, OOPSLA'86. 1986.

About the authors

Dilip Patel is head of the Centre for Information and Organisation Studies and Professor of Information Systems at South Bank University, UK. His main research interests include object technology, organisation theory and databases. He can be reached at dilip@sbu.ac.uk

Shushma Patel is a Principal Lecturer in Information Systems at South Bank University, UK. Her main research interests include Object technology and Information Systems.

Paul Schleifer undertook his PhD within the Centre for Information and Organisation Studies and is currently working in industry.