Behaviour Specification in Database Interoperation

Mark W.W. Vermeer and Peter M.G. Apers

Centre for Telematics and Information Technology
University of Twente, Enschede, The Netherlands
{vermeer,apers}@cs.utwente.nl

Abstract. We discuss the impact of locally implemented behaviour in a federation of object-oriented databases. In particular, given a specification of an integrated view of a number of component databases, we discuss the process of determining the global methods that are implicitly implemented by a given set of local methods on these component databases. To this end, we develop the notions of *objectivity* and *subjectivity* of local methods, indicating whether the execution of a local method affects the global view exactly as it affects the local database, *behaviour equivalences* between local methods, indicating whether local methods of different components have similar effect, and *behaviour concurrences*, indicating whether local methods respond to the same event.

1 Introduction

So far, database interoperation research has focused on the structural aspects of data integration. Even though the use of an object-oriented data model as the canonical model for interoperation has been widely advocated [6], attention for the extended structural modelling capabilities of such models has overshadowed their behavioural aspects. That is, object-oriented multidatabase management systems generally do not present object methods other than those implementing generic query and transaction facilities to a global user, in spite of the fact that component databases may have implemented application-specific methods with their local objects. In this paper, we investigate to what extent such locally defined methods can be incorporated in the global object view definition. It would be attractive to offer global applications a global method interface with comparable functionality. Such global methods, however, are virtual in the sense that they are implemented by calling the appropriate local methods at (multiple) component databases.

1.1 Approaches to behaviour in database interoperation

Many types of research can be regarded as somehow addressing behaviour in the context of database interoperation. We discuss three of them here.

In [3], Bertino et al. distinguished between structural and operational mappings. It was argued that operational mapping is a good alternative if a structural mapping cannot be achieved, for example if one of the systems to be integrated is not a DBMS. A global user is then presented with a set of operations rather than an integrated schema. In contrast, in this paper we discuss behavioural issues that arise once a structural integration has been performed.

Another idea is to use semantical information provided by method definitions to guide the process of schema integration [8]. Although this is an interesting

approach, in this paper we assume an integrated view has been defined, and consider the resulting global impact of local methods.

Behaviour sharing [5] occurs when a remote component offers additional services for local objects. For example, a remote method `PostScriptDoc.Display()` is executed on a local object o:`VLDBPaper`. The attention for semantical issues is restricted in that discrepancies among overlapping types and object sets are not considered. Moreover, no specification of methods other than their signature is considered.

1.2 Our approach: Method reuse

In our approach, integration of methods is treated using exactly the paradigm that is usual in structural database integration [7]. That is, a set of methods defined on the component data structures is assumed; these methods have been implemented autonomously and cannot be changed. We then concentrate on the following question: *Given a definition of an integrated view of a set of interoperable databases, each of which is equipped with a set of local methods, what is the set of methods applicable to the integrated view?*

We show how local behaviour specifications can be adapted to suit the global level through a process called *conformation* (Section 3), and discuss the applicability of local methods at the global level using the notions of *objective* vs. *subjective* local methods (Section 4). Subsequently, in Section 5 we use the notion of *behaviour equivalence* to determine globally applicable methods. In Section 6 we then introduce the idea of *behaviour concurrence* to express the fact that different methods may respond to the same event. As a context for discussion, we use the database interoperation methodology first described in [9], and summarised in Section 2. We believe that the relevance of the topics discussed in this paper goes beyond the specific methodology used here, however.

We use the behaviour specification possibilities offered by the object-oriented database specification language TM [1]. In TM, methods are specified in a functional manner, using a computationally complete data manipulation language. We assume a fully-fledged TM-specification of the interoperable component databases exists. Since database interoperation is concerned with so-called legacy systems, we need to obtain TM-specifications from existing behaviour implementations through *reverse engineering*. This was the subject of our previous work [11].

2 Structural Integration

We discuss structural integration in the context of the following example that is used throughout this paper.

Example: Joint concessions We consider the case of two oil companies `Comp1` and `Comp2`, that are about to create a joint venture to coordinate their exploitation of a set of oil fields in a certain region. Some of the concessions to exploit fields in this area are already shared by the companies. To implement the coordination, an integrated view of both parties' well databases, in which information about oil wells and their production is kept, is built. `Comp1`'s database is defined as follows (we list only data structures here; methods are introduced throughout upcoming sections).

```
Class Field
attributes   name          : string   estsize   : real    value     : real
             expproduction : real     wells     : PWell
end Field
```

```
Class Owner
attributes   name          : string    fields      : PField
end Owner

Class Well
attributes   name          : string    cost        : real      location    : Point
             type          : string    nrholes     : integer   lifetime    : integer
end Well

Class Point
attributes   x             : real      y           : real
end Point
```

This database is to be virtually integrated with the following database from Comp2. To a large degree, the databases are similar, but naming conventions and structural representations differ slightly. Moreover, this database describes only production wells, whereas Comp1 describes wells in general.

```
Class Concession
attributes   name          : string    price       : real      expproduction : real
             estsize       : real      quality     : integer   wells         : PProdWell
end Concession

Class ProdWell
attributes   name          : string    currproduction : real   x             : real
             y             : real      cost        : real      nrholes       : integer
end ProdWell

Class ConcHolder
attributes   name          : string    concessions : PConcession
end ConcHolder
```

2.1 Specification of integration

We assume that before behaviour integration is performed, a structural integration has been defined. Here we use our integration specification methodology from [9]. This methodology is instance-based in that it considers objects rather than classes to be an appropriate unit of integration. In short, the motivation for our approach is the argument that in absence of a common semantical context, it is more feasible for disparate sources to agree on relationships among the specific real-world objects that they describe, than to agree on the semantics of possible classifications for those objects.

The methodology requires the specification of *object comparison rules* and *property equivalence assertions*. Object comparison rules define conditions under which any of the following relationships hold between a remote object O' and a local object O or class C:

- **Identity**. O and O' represent the same real world object. This is represented as $Eq(O', O)$.
- **Strict similarity**. O' would locally be classified under C. This is represented as $Sim(O', C)$.
- **Approximate similarity**. Locally $C \cup \{O'\}$ would be considered a meaningful, more general class C''. This is represented as $Sim(O', C, C'')$.
- **Descriptivity**. Locally O' is considered a set of values S describing an object O'' which is identical to a local object O or similar to a local class C. This is represented as $Eq(O', O.S)$ or $Sim(O', C.S)$.
- **Constituency**. Locally O' is seen as a constituent of O; O and O' describe different levels of aggregation. This is represented as $Aggr(O, O')$.

Property equivalence assertions describe description overlap in local and remote types of related objects. These assertions are of the form $propeq(C.p, C'.p', cf, cf', df)$, where:

- p, p' are basic or derived local and remote properties, respectively,
- cf, cf' are *conversion functions* mapping the domains of p and p' to a common domain D, and
- $df : D \times D \rightarrow D$ is a *decision function* which determines a global value for the property given possibly different local and remote values. We require that for each decision function df, $\forall a \in D | df(a, a) = a$. In our view, functions such as *sum* used e.g. in [4] define derived global properties rather than determining values for equivalent local and remote properties.

The structural integration specification for our example is given below. It is not intended to illustrate structural integration is full, for a more complete discussion refer to [9]. We use predefined conversion functions such as *id*, the identity function, and decision functions such as *trust*, which assigns a specific database as the primary source for a property's value.

Eq(*O*:`Field`, *O'*:`Concession`) ← *O*.name = *O'*.name
Sim(*O'*:`Concession`,`Field`)
Eq(*O*:`Owner`, *O'*:`ConcHolder`) ← *O*.name=*O'*.name
Sim(*O'*:`ConcHolder`,`Owner`)
Eq(*O*:`Well`, *O'*:`ProdWell`) ← *O*.location.x=$_{eq}$ *O'*.x ∧*O*.location.y=$_{eq}$ *O'*.y
Sim(*O*:`Well`, `ProdWell`) ← *O*.type = 'production'
Sim(*O'*:`ProdWell`, `Well`)
Eq(*O*:`Point`,*O'*:`ProdWell`.{x,y}) ← *O*.x=$_{eq}$ *O'*.x ∧*O*.y=$_{eq}$ *O'*.y
Sim(*O*:`Point`,*O'*:`ProdWell`.{x,y})

propeq(`Field`.name, `Concession`.name, *id*, *id*, *any*)
propeq(`Field`.expproduction, `Concession`.expproduction, *id*, *BarrelsToGallons*, *avg*)
propeq(`Field`.value, `Concession`.price, *id*, *DollarsToPounds*, *any*)
propeq(`Field`.estsize, `Concession`.estsize, *id*, *id*, *trust*(`Comp1`))
propeq(`Owner`.fields, `ConcHolder`.concessions, *id*, *id*, *union*)
propeq(`Owner`.name, `ConcHolder`.name, *id*, *id*, *any*)
propeq(`Well`.name, `Prodwell`.name, *id*, *id*, *any*)
propeq(`Well`.location, `ProdWell`.(x,y), *id*, *CoordConvert*, *any*)
propeq(`Well`.cost, `ProdWell`.cost, *id*, *DollarsToPounds*, *avg*)
propeq(`Well`.nrholes, `ProdWell`.nrholes, *id*, *id*, *any*)

where $=_{eq}$ is defined as equality modulo domain conversion. As indicated by the specification, it is assumed that `Well` and `ProdWell` that have the same location (modulo a coordinate conversion to account for the different coordinate systems used by `Comp1` and `Comp2`) represent the same real world object. Moreover, `Well` objects whose type is 'production' are regarded to be strictly similar to `ProdWell`-objects of `Comp2`.

2.2 Conformation and merging

As a result of a specification as defined above, an integrated or global view of the local and the remote database can be constructed. This construction is a two-step process of *conformation* and *merging* analogous to the two steps distinguished for schema integration in [2]. Our discussion here is necessarily brief; the interested reader is referred to [9].

Conformation In the conformation step, the local and remote database are brought into a common semantical context, so that they can be merged. This involves the settling of object-value conflicts resulting from descriptivity relations between objects. This is done by creating virtual objects from values and/or

casting objects into property values describing other objects. In our example, the description of a well location as an (x,y) value pair describing a well or as a separate `Point` object must be conformed. We here assume that this is done by creating virtual `VirtPoint`-objects from the ⟨ x,y ⟩ values of `ProdWell`.

Equivalent local and remote properties p and p' are turned into *conforming properties* p_c and p'_c by assigning them identical names and converting them to identical domains. Examples include the renaming of 'price' to 'value', the conversion of production figures to a common unit, and the choice for a common coordinate system to describe well locations.

Merging In the merging step, objects between which an equivalence relationship has been determined, are merged into a single global object. Equivalent properties are merged into an integrated property and assigned to the integrated class hierarchy. Moreover, the value of global properties is determined from the conformed local and remote ones, using a decision function where applicable.

3 Conformation of Method Specifications

The two phases of conformation and merging are applicable to the reuse approach to method integration as well. In this section we discuss the conformation of method specifications. A *conformed method specification* is a description of locally implemented functionality in global terms (see Figure 1). Note that a conformed method specification itself is not implemented directly, but can be executed by calling the locally implemented method.

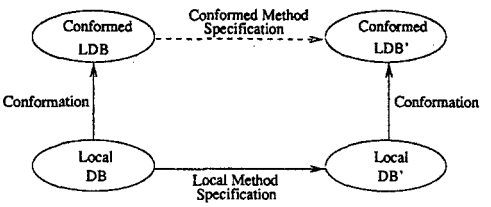

Fig. 1. Method conformation

3.1 Transformation types

We distinguish two types of transformations applied during the conformation phase.

- *Structural transformations* σ are used to resolve objects versus value conflicts and naming conflicts. Given a property p in the local database, $\sigma(p)$ returns the corresponding property in the conformed database.
- *Value transformations* ν are used to conform the domain of equivalent properties. Given a value x of a property p in the local database, $\nu_p(x)$ returns the corresponding value of $\sigma(p)$.

Example Consider the property **ProdWell.x**. To conform the use of position coordinates, the x and y values of **ProdWell** are transformed into separate **Virtpoint**-objects. Hence σ(**ProdWell.x**)=**ProdWell**.location.x. Moreover, coordinate values of **ProdWell** are converted to account for the different coordinate systems used by **Comp1** and **Comp2** using the function $\nu_{\texttt{ProdWell.x}} = \mathit{CoordConvert}$. Thus, $\mathit{CoordConvert}$(**ProdWell.x**) returns the value of the conformed property **ProdWell**.location.x. □

3.2 Obtaining conformed method specifications

To obtain conformed method specifications, we first note that local methods themselves are value transformations, i.e. local methods do not change the structure of the database.

Given a method specification $\delta : DB \rightarrow DB'$, its conformed form is a mapping $\Delta : \sigma(DB) \rightarrow \sigma(DB')$. It is obtained from the specification of δ as follows.

1. An lhs-reference to a property p is replaced by $\nu_p^{-1}(\sigma(p))$.
2. An rhs-reference (assignment) of a value x to a property p is replaced by an assignment of $\nu_p(x)$ to $\sigma(p)$.

See also Figure 2. Typically, specifications thus obtained can be rewritten to more elegant ones using distributive and other properties of ν_p w.r.t. δ.

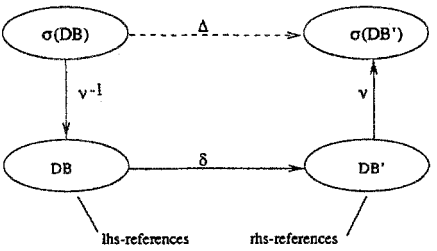

Fig. 2. lhs and rhs conformation

Example Consider the following method specification.

object update method for Concession
Depreciate(**in** p: real)=
  if price>p
  then self except price=price-p
  else self except price=0
  endif

This method acts upon the 'price' property of the **Concession**-objects in the local database state only. According to the structural integration specification, the following transformations in the conformation phase are relevant to this method specification: (1) σ here consists of the renaming of the property 'price' to 'value' (a structural transformation) ; (2) ν_{price} is the conversion function *DollarsToPounds*. Thus, the conformed method specification describing the transformation Δ would be

object update method for Concession
Depreciate(**in** p: real)=
  if *PoundsToDollars*(value)>p

then self except value=*DollarsToPounds(PoundsToDollars*(value)-p)
else self except value=*DollarsToPounds*(0)
endif

This specification can be simplified using the distribution property of *DollarsToPounds* over subtraction and the knowledge that *DollarsToPounds* preserves order. The specification can be rewritten, yielding:

object update method for `Concession`
Depreciate(**in** p: real)=
if value>*DollarsToPounds*(p)
then self except value=value-*DollarsToPounds*(p)
else self except value=0
endif

□

4 Objectivity and Subjectivity

4.1 Applicability of local methods

Even though expressed in global terms, a conformed method specification is not necessarily a correct specification of the result of the execution of a local method on the global state.

Example As a simple example, consider the local method

object update method for `Field`
NewResources(**in** am: real)=
self except expproduction=expproduction + am

Conformation leaves the specification of this method unaffected. However, if this method is applied to a local object O, the state of the global object $\dot{O}$ representing O is not affected accordingly, as according to the structural integration specification there may exist a remote object O' : `Concession` such that $Eq(O, O')$. The 'expproduction'-value of $\dot{O}$ is then defined as the average of the 'expproduction'-values of O and O'. Hence an increase of the 'expproduction'-value of O by am leads to a corresponding increase in the 'expproduction'-value of $\dot{O}$ by only $am/2$. □

In [10], we introduced the notions of *objectivity* versus *subjectivity* in the context of database interoperation. We here elaborate on this subject in the context of methods. In our terminology, effects as the one above are due to the *subjectivity* of the property that the local method is defined upon. A database modelling assertion, such as a property value, is called objective iff its validity is independent of the implicit assumptions made within the context of a particular database; otherwise it is called subjective.

4.2 Objectivity of property values

We say that a property value v associated with a property p of a local object O is *objective* iff either:

1. O is not involved in an equivalence relationship with a remote object O'; or
2. O is involved in an equivalence relationship with a remote object O', but no property equivalence assertion for p has been defined; or

3. If O is involved in an equivalence relationship with a remote object O', and a property equivalence assertion has been defined for p, we must distinguish between the following subcases according to the type of decision function involved:
   (a) *Conflict ignoring function*
   This represents the situation where the decision function does not deal with possible value conflicts. That is, non-deterministically any of the values is chosen (denoted in the example by the *any* function). In this case, v is objective. Thus, `Concession.price` and `Field.value` hold objective values in our example specification.
   (b) *Conflict avoiding function*
   Here one of the equivalent properties is chosen as the most reliable source of values for the integrated property (the function *trust* in our specification). v is objective iff it is a trusted property value. Thus, `Field.estsize` values are objective in our example specification, but `Concession.estsize` values are not.
   (c) *Conflict settling function*
   The conflict is settled by picking one of the values using a certain decision procedure. Examples of such functions are *max* and *min*. v is objective iff it meets the criteria of the decision function.

We will also use a stronger notion of objectivity for property values. A value v of a property p is called *strictly objective* iff condition 1,2, or 3b above hold. A strictly objective value will remain objective after an update, as discussed in Section 4.4.

4.3 Objectivity of class extensions

A class C is said to have *objective extension* iff the global extension of C is identical to the local extension, i.e. $\nexists O' : Sim(O', C)$. Note that in this definition, objectivity of local class extensions may be affected by the addition or deletion of remote objects. Hence we also define a stronger notion of extension objectivity. A class C has *strictly objective extension* iff no object comparison rules are defined on C.

Objectivity of class extensions affects the objectivity of class methods as shown in the next subsection.

4.4 Objectivity of methods

An objective method is a method that has the same effect globally as it has locally. As shown in the previous example, objectivity of a method is related to the objectivity of the properties it operates on. However, in the case of update methods, additional factors play a role. Updates may establish and/or break object relationships, thus effecting the global view beyond their specification.

Example Consider the conformed object update method

```
object update method for Well
ChangeLocation(in newx, newy : real)=
  self except(location.x= newx,
              location.y= newy)
```

Suppose this method is executed against an object O :`Well` with parameters nx, ny. Assume that initially there does not exist an object O':`ProdWell` such that $Eq(O, O')$. Suppose however, that there does exist a conformed remote object O' :`ProdWell` where

O'.location.x= nx, and O'.location.y= ny. Hence after execution of ChangeLocation, $Eq(O, O')$ holds. Thus, in the global view, O and O' are now represented by a single global object $\dot{O}$. Now consider the property value O'.cost=c'. The global value of $\dot{O}$.cost is now suddenly $avg(c, c')$, where $c = O$.cost. Hence as a side effect of this update, at the global view the 'cost' property may change as well, although this is not specified by the method body. Therefore the method is subjective. □

The crux of this example is that for update methods to be objective, the state *before* and *after* the application of the method must be objective. This is the motivation for the introduction of strict objectivity of property values. Strict objectivity of class extensions is motivated by a similar effect for class update methods: the creation of new objects or the deletion of old ones may not be effectuated in the global view.

Example Consider the class update method

class update method for `ProdWell`
CloseWell(**in** wellname : string)=
  self minus
    collect x **for** x **in self**
    iff x.name= wellname

Calling this method with parameter "North123" intends to remove a well called "North123" from class `ProdWell`. However, suppose that this particular well is represented both in class `Well` as an object O (where O.type='production') and in `ProdWell` as an object O'. Hence at the global level it is represented by an object $\dot{O}$ of both class `Well` and `ProdWell`. Upon deletion of O' from `ProdWell`, $\dot{O}$ is *not* deleted from `ProdWell`, since O is now similar to `ProdWell`, as specified by the object comparison rules. □

These examples motivate the following definition. Let P_M be the set of properties addressed by a method M. Furthermore, let PC_O be the set of properties involved in comparison rules on O. Objectivity of M is defined as follows:

- An *object retrieval method* M on an object O is called objective iff each $O.p$, where $p \in P_M$, has objective value.
- An *object update method* M on an object O is called objective iff each $O.p$, where $p \in P_M$, has strictly objective value, and M does not update any of the properties in PC_O.
- A *class retrieval method* M on a class C is called objective iff each $O.p$, where $O \in C$ and $p \in P_M$, has objective value, and C has objective extension.
- A *class update method* M on a class C is called objective iff
  - M is of **self union** or **self minus** type and C has strictly objective extension.
  - M is of **self except** type and each $O.p$, where $O \in C$ and $p \in P_M$, has strictly objective value, and M does not update any of the properties in PC_O, and C has objective extension.

5 Global Application of Local Methods

Equipped with the notions of objectivity and subjectivity of local methods, we now turn to the derivation of method specifications on the global view from a given set of conformed local method specifications.

5.1 Objective methods

Objective methods have global effects as specified by their conformed specification; hence they can be seen as methods on the global view.

Example Consider the local method

object update method for `Field`
SizeEstUpd(**in** am: real)=
  self except estsize=estsize + am

As the decision function defined for 'estsize' is *trust(DB1)*, this is an objective object method for any O :`Field`. Hence this method specification can be reused at the global level; when executed locally, the state of the global view changes exactly as described by this specification. □

5.2 Subjective methods

By definition, a subjective method specification M cannot directly be reused at the global level. We distinguish between update and retrieval methods here.

Subjective retrieval methods Any conformed subjective retrieval method M can be implemented at the global level through what is known as *materialisation*. To implement a subjective local retrieval method M, the global state of P_M is materialised as specified by the object comparison rules and property equivalences, and then M is evaluated against this materialised state. Two assumptions are implicit here:

1. The state of component databases can be accessed entirely. Note that strictly speaking, this assumption is not in accordance with the *strict reuse* approach, as it is assumed that a component database can be accessed through its predefined methods only. This may be relaxed to what we might call *extended reuse*, where it is assumed that the state of any local object can be accessed through implicit *get-value* operations for each of its attributes.
2. The functionality specified by M can be implemented at the global level. Hence fully-fledged method evaluation must be available at the global level.

Thus, if these conditions are satisfied, any subjective retrieval method M is applicable at the global level. Otherwise, M may be effectuated at the global level by distribution over the components, which is the standard approach for dealing with subjective update methods, as discussed in the next subsection.

Subjective update methods To implement a subjective local update method M at the global level, two conditions must be satisfied:

1. The behaviour defined by M must be *distributable* over the decision functions defined for P_M.
2. *Equivalent behaviour* M' must be available at other components.

As to the first condition, whether a given method M is distributable depends on whether its specification distributes over the decision functions defined on P_M.

Example Consider the subjective (retrieval) method

object retrieval method for `Owner`
AvgValue(**out** real)=
  avg fields **over** value

The decision functions *union* and *avg* defined for 'fields' and 'value', respectively, prevent this method from being calculated in a distributed manner. □

Although a proof tool for deciding upon the distributivity of a given method over the decision functions defined on its properties might be achievable due to the formal semantics of TM, we here assume that a designer specifies distributivity properties of subjective method specifications when designing the integrated view.

In any case, methods involving updates on properties that occur in object comparison rules cannot be distributed, due to the side-effects described in the previous section.

As to the second condition, the question whether equivalent behaviour M' exists in a component database is related to the level of autonomy of component databases w.r.t. behaviour definition. In a *re-engineering* context, we could simply define such a method M' on the remote database. This simple solution obviously violates the autonomy of the remote database. The reuse approach to behaviour integration, however, requires that M' is implementable in terms of the existing remote methods.

Behaviour equivalences That is, we need to find *behaviour equivalences* of the form $M' \equiv e$ such that e is an expression that can be evaluated at the method interface offered by the conformed component database. The type of behaviour equivalence expressions allowed depends on the type of reuse approach, as distinguished above.

- *Strict reuse* would allow only method calls to occur in such expressions, with possibly complex parameter specifications. Hence no additional processing beyond that defined by the remote methods is allowed.
- Since in the *extended reuse* approach we may access the entire remote object state, lhs-type property references and method calls are allowed to occur in such expressions.

Determining behaviour equivalences should in principle be automatable through matching of conformed local and remote method specifications. A more pragmatic approach would be to have users suggest behaviour equivalences, which can then be checked for validity. Due to space limitations, we present a brief example only.

Example A simple behaviour equivalence that may be specified is
`Field`.Depreciate(p) ≡ `Concession`.Depreciate(*PoundsToDollars*(value*p))
Moreover, the (subjective) Depreciate-method is distributable over the decision function *any*. Together this makes Depreciate a globally applicable method.

When distributing subjective *class methods*, we have to take object comparison conditions into account. Consider for example the local method

class update method for `ProdWell`
AddCosts(**in** p: real, n: integer)=
  replace (x **except** cost=cost*p)
        for x **in** self
        iff x.nrholes > n

This is a subjective method due to the subjectivity of the class extension. This method is distributable over the decision functions *avg* and *any* defined on 'cost' and 'nrholes', respectively. Suppose now that we have a component *object* update method

object update method for `Well`
AddCosts(**in** p: real)=
  self except cost=cost*p

Then we have the behaviour equivalence

`ProdWell`.AddCosts(p,n) ≡ **replace** x.AddCosts(p)
for x **in** `Well`
iff x.nrholes>n **and** x.type = 'production'

Note the addition of the object comparison condition, and the use of `Well.AddCosts` as a subroutine of `ProdWell`.AddCosts. Note furthermore that this expression is not allowed under the strict reuse approach, as it requires the retrieval of local state. □

6 From Globally Applicable Methods to Global Methods

Typically, globally applicable methods derived from different component databases are not independent from a global perspective. In particular, such methods may have been designed as a response to the same or (temporally) related events. Hence we need to coordinate the global execution of such methods. Temporal and logical coordination of activities performed in different systems is a subject of *workflow management* research; this is not discussed here. The case where global methods define reactions on the *same* events is of special interest with respect to our discussion so far, however. Global methods related in this way are called *concurrent* here. They should be combined into a single global method.

Example Consider the objective methods

object update method for `ProdWell`
StartUp()=
  self except currproduction = 0

object update method for `Well`
StartUp()=
  self except lifetime = 0

These methods have been designed in response to the same event, viz. the startup of a (production) well. Both are applicable at the integrated level. From a global point of view, they should be executed jointly whenever this is possible. That is, objects appearing in both `ProdWell` and `Well` should have a single StartUp method, specifying both startup actions. □

Note the difference between equivalent and concurrent behaviour. Equivalent behaviour, introduced in the previous section, occurs when identical actions are defined. Concurrent behaviour is defined in response to identical events, but the actual actions taken may be different.

Simply merging the execution of both concurrent methods is not the only way to combine such methods into a single one. An important alternative is the principle of *overriding*. Given methods M_1 and M_2 responding to the same event, we say that M_2 overrides M_1 if M_1 is defined to be applicable only if M_2 is not.

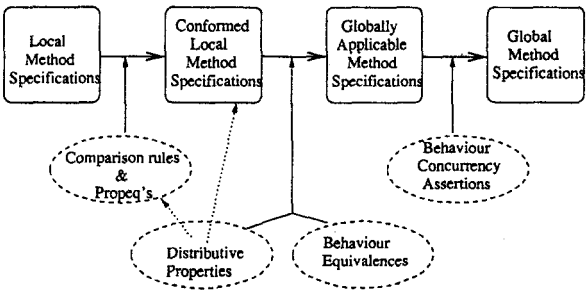

Fig. 3. Overview of global behaviour specification

Example Consider the methods

object retrieval method for `Well`
Expensive(**out** boolean)=
  cost/lifetime > EPSILON

object retrieval method for `ProdWell`
Expensive(**out** boolean)=
  currproduction*OILPRICE < cost

Assume these subjective methods are both globally applicable using materialisation. Here the second definition of Expensive should override the first one. That is, objects appearing in both `ProdWell` and `Well` should have a single Expensive method: the one defined in `ProdWell`. □

Although in principle we might allow any combination of concurrent global methods into a single one, these two cases are the most relevant ones.

Note that whereas distributive properties and behaviour equivalences can in principle be determined from the conformed method specifications, behaviour concurrences must be user-specified. Thus, to integrate globally applicable methods into global methods, we need *behaviour concurrency assertions* of the form *behconc*(M, M', [MERGE|OVERRIDE]), specifying that M' should either override or be merged with M at the global level, provided that both are globally applicable.

7 Discussion

Figure 3 gives an overview of behaviour specification in database interoperation as it has been discussed in this paper.

We have shown how *conformation* of local method specifications accounts for the reconciliation of representation differences between the sites in the specification of locally implemented behaviour. This allows the comparison of methods implemented at different sites. Subsequently we introduced the notions of *objectivity* and *subjectivity* of local methods. Only objective methods have direct global applicability; subjective local methods have global applicability only if *equivalent* behaviour exists or can be introduced at other sites, and the behaviour is *distributable*. We have shown that there is a relationship between subjectivity of properties and subjectivity of methods. Finally, we noted that

methods, although different in specification, may have been designed as a response to the same logical event. We illustrated two principle ways of dealing with such *concurrent* methods.

We may conclude that the specification of behaviour in the context of database interoperability is an engineering activity, that can be clearly structured and supported by tools following the principles discussed in this paper.

References

[1] H. Balsters, R. A. de By & R. Zicari, "Typed sets as a basis for object-oriented database schemas," in *Proceedings Seventh European Conference on Object-Oriented Programming, July 26–30, 1993, Kaiserslautern, Germany, LNCS #707*, O. M. Nierstrasz, ed., Springer–Verlag, New York–Heidelberg–Berlin, 1993, 161–184, See also `http://wwwis.cs.utwente.nl:8080/oodm.html`.

[2] C. Batini, M. Lenzerini & S. B. Navathe, "A comparative analysis of methodologies for database schema integration," *ACM Computing Surveys* 18 (December 1986).

[3] E. Bertino, M. Negri, G. Pelagatti & L. Sbattella, "Integration of heterogeneous database applications through an object-oriented interface," *Information Systems* 14 (1989), 407–420.

[4] U. Dayal & H-Y. Hwang, "View definition and generalization for database integration in a multidatabase system," *IEEE Transactions on Software Engineering* 10 (November 1984), 628–645.

[5] D. Fang, S. Ghandeharizadeh, D. McLeod & A. Si, "The design, implementation, and evaluation of an object-based sharing mechanism for federated database systems," in *Proceedings Ninth International Conference on Data Engineering, Vienna, Austria, April 19–23, 1993*, IEEE Computer Society Press, Washington, DC, 1993, 467–475.

[6] E. Pitoura, O. Bukhres & A. Elmagarmid, "Object orientation in multidatabase systems," *Computing Surveys* 27 (June 1995), 141–195.

[7] A. P. Sheth & J. A. Larson, "Federated database systems for managing distributed, heterogeneous and autonomous databases," *ACM Computing Surveys* 22 (September 1990), 183–236.

[8] C. Thieme & A. Siebes, "Guiding schema integration by behavioural information," *Information Systems* 20 (1995), 305–316.

[9] M. W. W. Vermeer & P. M. G. Apers, "On the applicability of schema integration techniques to database interoperation," in *Proceedings Fifteenth International Conference on Conceptual Modelling (ER'96), Cottbus, Germany*, Springer-Verlag, Berlin, 1996.

[10] M. W. W. Vermeer & P. M. G. Apers, "The role of integrity constraints in database interoperation," in *Proceedings 22nd International Conference on Very Large Databases (VLDB'96), Bombay, India*, Morgan Kaufmann Publishers, San Mateo, CA, 1996, 425–435.

[11] M. W. W. Vermeer & P. M. G. Apers, "Reverse engineering of relational database applications," in *Proceedings Fourteenth International Conference on Object-Oriented and Entity-Relationship Modeling (ER'95), Gold Coast, Australia*, M. P. Papazoglou, ed., Springer-Verlag, New York–Heidelberg–Berlin, December 1995, 89–100, LNCS #1021.