4

Integrity Constraints in Federated Databases

Martin S Olivier
Department of Computer Science, Rand Afrikaans University
PO Box 524, Auckland Park, Johannesburg, 2006 South Africa
Email: molivier@rkw.rau.ac.za

Abstract

This paper considers integrity issues that arise because of the relative autonomous nature of sites in a federated database. Such autonomy requires guarantees from the provider of a service and enforcement of such guarantees. Guarantees are expressed as integrity constraints.

Federated databases can form the basis of many forms of automated interorganisational processes. Successful automation of such processes depends on conformance to expectations of all organisations involved. Although mechanisms (such as contract law) exist to ensure compliance, the existing mechanisms are too complex to automate and also too lengthy for many time-critical processes or transactions.

The paper describes an approach where a service provider provides a guarantee, an analysis site simplifies the checking of such constraints and a certification site certifies that guarantees will be enforced.

Keywords

Keyword codes: C.2.4, H.2.5, K.6.5
Keywords: Distributed Systems, Heterogeneous Databases, Security and Protection

1 INTRODUCTION

Interorganisational sharing of information is accomplished by either sending the shared information via a network to the recipient, or by supplying the recipient with a reference where the information can be accessed. EDI illustrates the first case. Here information is 'packaged' and sent to the recipient, after which it occurs on both the sender's and recipient's machines. The World Wide Web illustrates the other form of sharing: A *hyperlink* 'points' to information on a Web site somewhere in the world and the recipient has to request the shared information. When requested, the information is supplied to the user, who uses it and then discards it; if required again, the information is simply requested again. This form of sharing is often used by organisations who give their partners and/or clients access to their databases, for example, suppliers who give retailers access to their

P. Samarati et al. (eds.), *Database Security*

catalogues, banks who give clients access to their bank accounts and governments who give citizens access to policy documents—all via networks.

We will refer to the first form of information sharing (where information is 'copied' to the recipient) as *sharing by value*, while the second form of sharing (where only a 'pointer' is supplied and the 'recipient' has the responsibility to retrieve it whenever required) will be referred to as *sharing by reference.*

In order to fulfil future requirements interorganisational hypermedia systems will have to incorporate database functionality. On the other hand shared databases will have to allow greater flexibility in sharing possibilities. Consider the Hyper-G (Kappe and Maurer, 1994) and Carnot (Woelk *et al*, 1995) projects as examples of the envisaged trends.

Apart from confidentiality and availability that has to be ensured in such databases, the integrity of such databases also needs attention: Referential integrity, for example, requires that an entity referenced by another entity has to exist (while it is referenced). However, if the referenced entity exists on another (autonomous) site than the referencing entity, this is not easily ensured. This paper considers sharing by reference in federated databases. In particular it addresses the situation where the integrity (or correct operation) of one database depends on assumptions about the state of other databases in the federation. Section 2 briefly considers existing work on integrity and security in federated databases. Section 3 motivates the need for constraints in federated databases further. Section 4 considers the nature of constraints in object-oriented databases, while section 5 considers the nature of constraints in federated databases. This is followed by the conclusion.

2 BACKGROUND

Integrity issues received much attention in the 1970s and early 1980s. However, despite this attention no consensus has been reached about the exact nature of integrity; neither have integrity mechanisms been widely implemented in commercial products.

See Eswaran *et al* (1976), Fernandez *et al* (1981:107–48), Date (1983:35–81), Sandhu and Jajodia (1991; 1995) and Abrams *et al* (1993) for discussions of integrity issues. The work described by these authors primarily considers integrity issues in centralised relational databases. However, the concepts also apply to other database models.

The class of distributed databases on which this paper focusses is that of federated databases. A *federated database* is a distributed database where the sites that form the distributed database have a high degree of site autonomy—that is, management of the distributed database is delegated to the sites as far as possible. See Özsu and Valduriez (1991) for a discussion of distributed databases and Özsu and Valduriez (1991:81,89) and Sheth and Larson (1990) for a discussion of federated databases in particular. See Thuraisingham (1994) for a discussion of the issues that must be considered when designing a secure federated database.

In this paper we are particularly interested in federated databases where the nodes are not owned by the same party and therefore subject to different security policies. See Olivier (1995) for an implementation strategy for such databases: the self-protecting object model (SPO). The SPO model is based on three components: The *trusted common core* (TCC) occurs on all nodes of the federated database and is trusted by all members of the federation. *Trusted extension* (TE) methods are associated with the entities in the database that are to be protected; whenever such an entity is accessed its associated TE

methods are activated to perform the required access control. Each site also has a *trusted local extension* (TLE) that supports the TE methods of that site by providing them with information about users, user groups and other information required for security purposes. The TLE and TE methods of a site only have to be trusted by the particular site (and not by other members of the federation).

As stated above, a TE method is activated when the entity protected by it is accessed. This notion corresponds to production rules (or triggers) found in active databases, but in SPO it is currently restricted to rolling back transactions when unauthorised accesses occur or, as used later in this paper, when constraints are violated. SPO also differs from current active databases because it is intended to provide trusted execution of these methods in an autonomous distributed environment. See Dayal *at al* (1995) for an introduction to active database systems.

The research in this paper assumes that the underlying databases are object-oriented. For the purposes of this paper we assume that an object-oriented database can be seen as a collection of objects accessible through the methods they provide. In order to access a service, the user formulates a request in the form of a message which identifies the method and supplies the additional information to enable the request to be handled. See Kim (1995) for a description of object-oriented databases and Lunt (1995), Olivier (1994) and Rabitti *et al* (1991) for a discussion of security in such databases.

For reasons of convenience we will sometimes refer to the methods of an object as *services* and to the site that owns the object with the concerned service as the *service provider.* The user who uses the service will be referred to as the *recipient.*

Rusinkiewicz *et al* (1991) have specified a declarative specification for interdatabase dependencies. To ensure that constraints are not violated these dependencies are considered when transactions are executed and new transactions are (recursively) spawned until system consistency is restored.

Constraint management in heterogeneous systems has been studied by Chawathe *et al* (1993; 1996). Their approach supports a more "relaxed" notion of integrity than ours (since constraints may be violated within given limits). In addition, their approach is intended for heterogeneous systems, while our approach assumes a common infrastructure at all nodes. More importantly, the Chawathe *et al* approach depends on an interface specification that has to "correctly reflect the actual behavior provided by the database containing that item" (Chawathe *et al*, 1993:9).

Ceri and Widom (1993) have also considered enforcement of constraints in loosely coupled distributed databases. Here consistence specifications are translated to production rules to be used at the distributed databases. Production rules can enqueue commands to be executed at remote nodes to effect consistency. Again the assumption is that a component database will (in all cases) activate the production rules installed on it when required.

The protocol by Ceri and Widom (1993) forms one of a family of protocols proposed by Grefen and Widom (1996). These protocols illustrate that alternative protocols for constraint checking in federated databases exist—each with its relative advantages and each with different requirements imposed on the component databases.

In all the work described above the assumption was made that the component databases can be trusted to handle their share of the constraint checking. Although such an assumption is realistic in a federation owned (or administered) by a single party, this assumption cannot necessarily be made if the nodes are owned by different parties—in particular, if

one such party can profit by avoiding some constraint checks (or rule invocations) on the local system. The current paper focusses on the situation where the component systems are owned and administered by different organisations and where mutual trust cannot be assumed, but has to be enforced (or at least verified).

Another interesting difference between all work on integrity in distributed databases referred to above and the current paper is the relative dynamic nature of constraints in the current paper: Constraints are (dynamically) requested by other nodes before related actions are initiated. In addition constraints expire at a given time or when a given event occurs. From the examples given later is it clear that the period of validity is an important aspect of many constraints.

3 MOTIVATION

As mentioned in the introduction, information can be shared by reference or by value. The primary reasons for sharing information by reference, rather than by value are

1. To allow updates to the information to be reflected in the receivers' 'copies' (or views) immediately, without redistribution of the the updated information;
2. To facilitate better access control—consider, for example, the case where a client pays for information accessed, but where the reference is provided for free;
3. To allow the client to update the shared information; and
4. To allow volumes of data to be shared that cannot be copied in practice—consider, for example, the volume of data (eventually) pointed to by almost any Web-site.

Sharing information by reference poses new problems: While information is often shared by reference because the information can be modified (see reasons 1 and 3 above), indiscriminate modification may adversely influence the recipients. Consider the following examples where the communicating parties are located at different nodes of a federated database.

1. A medical record, at one site, on which a specialist, at another site, bases a diagnosis is later modified so that the diagnosis becomes unacceptable.
2. An advertisement quotes a price, but the price is increased after an order has been placed.
3. A telephone directory refers to another directory (kept on another site) for numbers in a given area. The telephone directory on the other site is subsequently deleted, leaving a 'dangling' reference in the first.

Although ad hoc solutions can be used to address the problems mentioned above, a general solution is required—a guarantee from the service provider that the information will not be modified in some specific ways, or, that the information will indeed be modified in some specific ways. Such a guarantee has to be valid for a given period, and has to be unforgeable by the recipient. This section motivates the need for such guarantees—especially in autonomous distributed systems. Apart from the fact that these guarantees are used in autonomous distributed systems, they are similar to integrity constraints used in the work referred to in the previous section.

Consider the following scenario to illustrate how such guarantees may be used. Assume that a general practitioner (GP) refers a patient to a specialist at a hospital. Assume that the GP prepares a referral letter on her local system (and includes it in the patient's file there). The specialist is given a reference to this referral letter (and possibly to the patient's file—see later). It is important for the specialist that this referral letter will not be modified after treatment has started. Assume REFLETTER refers to the referral letter and GETTEXT is the method used by the specialist to obtain the contents of this letter. The fact that the contents cannot be changed by any message sent to it (from the point of referral) can be expressed as:

REFLETTER:GETTEXT = REFLETTER':GETTEXT

Here REFLETTER' refers to the object before a message is received, while REFLETTER refers to it after the message has been handled.

When admitting the patient to hospital, the specialist may submit details to the patient's medical aid for approval. In order to approve expenses, the medical aid may specify that the total account may not exceed \$10 000 and that the daily fee may not be increased after approval. This can be specified as follows:

ACCOUNT.TOTAL $\leq$ 10 000

ACCOUNT.DAILYFEE $\leq$ ACCOUNT'.DAILYFEE

If the specialist refers the patient to a dietician at some point, and a diet is prescribed for the patient, the specialist can either expect that the prescribed diet will not be modified

DIET.GETTEXT = DIET'.GETTEXT

or that, whether the dietician modifies the diet or not, that the diet must be available, specified as:

DIET.GETTEXT = ?

Referring to the patient's file on the GP's system, the specialist may also require that the existing entries in the file cannot be modified (but new entries may be appended). This may be specified as follows:

PREFIX(PATFILE':GETTEXT, PATFILE:GETTEXT)

In section 4.1 the notation used to express constraints in this paper will be considered in more detail.

4 CONSTRAINTS IN OBJECT-ORIENTED DATABASES

The challenge of consistency constraints is to explicitly specify (and enforce) those assumptions made about the contents of the database on which correct operation of the database depends. In the case of relational databases, integrity restrictions are specified in terms of the contents of tables. In the object-oriented case such restrictions can usually be encoded as part of the normal code encapsulated in the object. Alternatively, con-

straints can be specified in a special notation and also be encapsulated as part of the object.

However, since the behaviour of an object is fully described by the methods it supports, it is also possible to specify integrity constraints in terms of methods. Such constraints may indeed be preferable above constraints embedded in objects. 'External' constraints are independent from the implementation of the object (and still apply if the implementation is changed). In fact, some of the assumptions made about some objects cannot be encapsulated as normal code: for example, the fact that a method M is only intended to be used by some other method M' of another object cannot be encoded in this way (compare views, Hailpern and Ossher, 1990; Shilling and Sweeney, 1989). This is precisely the type of constraint we are interested in in this paper: those constraints that are expressed 'outside' the object, either because they cannot reasonably be included as code of the object or because some other reason motivates an external constraint.

Since public methods present the only manner in which an object can be accessed from the 'outside' it follows that external constraints have to be specified in terms of messages. We consider the nature of external constraints before we consider the enforcement of such constraints.

4.1 External constraints

Two categories of external constraints have to be considered: those limiting the use of methods (or, effectively, the sending of messages) and those limiting changes to the state of objects. Constraints in the first category may be expressed as follows:

PREVENT *message(s)* UNTIL *condition*

The following examples serve to illustrate such constraints. They refer to the examples given in a previous section.

PREVENT ACCOUNT:INCREASEDAILYFEE UNTIL PATIENT:RELEASE

PREVENT REFLETTER:SETTEXT UNTIL 1 JANUARY 1998

PREVENT ADVERTISEMENT:INCREASEPRICE,
ADVERTISEMENT:SETPRICE UNTIL TODAY+14 DAYS

PREVENT DIRECTORY:DELETE UNTIL CONSTRAINT DROPPED

See Bertino *et al* (1996) for a model to enforce temporal authorisation.

Constraints intended to limit the changes to the state will also be expressed in terms of messages:

VERIFY *conditional expression* UNTIL *condition*

To illustrate this class of constraints consider the following. The first two constraints have

the same intention as those given as PREVENT constraints above.

VERIFY ACCOUNT.DAILYFEE = ACCOUNT'.DAILYFEE UNTIL PATIENT:RELEASE

VERIFY REFLETTER:GETTEXT = REFLETTER':GETTEXT UNTIL 1 JANUARY 1998

VERIFY DIRECTORY:EXIST

VERIFY ACCOUNT.TOTAL $\leq$ 10 000

Note that the intention of specifying such a constraint using messages has the meaning "If the message(s) were to be sent, the value(s) returned should fall within the specified limits"; a message that will not cause state changes can be sent to check the constraint; if a message is sent only to check a constraint and it changes the state, the effects of this message will have to be rolled back. Enforcement of constraints will be discussed in section 4.3.

For PREVENT constraints it is possible to add a clause of the form

BY *subject(s)*

which explicitly specifies the subjects currently prevented from sending the given methods.

Additionally both types of constraint can have a clause which specifies what should happen if the constraint is violated. So far we have assumed that the message is simply rejected. The only other possibility considered in this paper is that of logging the fact that a constraint has been violated. More alternatives do exist but fall outside the scope of this paper.

In most situations VERIFY constraints are preferable over PREVENT constraints: A complex object may support many mechanisms to obtain the same result. Consider, for example, a supplier who determines the price of an object from cost price (in some foreign currency), the current exchange rate, the supplier's own profit margin, volume discounts given, value added or sales tax applicable, as well as other factors. If the supplier guarantees that a price will not be increased for some period, it is easier to VERIFY the price than to PREVENT changes to all factors mentioned. (And note that it is possible that the exchange rate can change together with the supplier's profit margin so that the sales price remains unaffected.)

4.2 Formalisation

Here the concepts described earlier will be formalised by using a mathematical model similar to that used by Fernandez *et al* (1981). Fernandez *et al* (1981) describe an integrity rule as a "tuple (O, t, c, p, ap), where O is the *data object* to which the rule applies, t indicates for which *access type* the rule will be invoked, c is a predicate expressing a *condition* that must be true in order for p to apply to O, p is an assertion (semantic constraint) that must be true for an occurrence of the object O, and ap is an auxiliary procedure that specifies what the system will do if p is not true."

For the purposes of this paper an integrity constraint is viewed as a tuple

$c = \langle m, p, s, e_{exp}, t_{start}, t_{exp}, a \rangle$

where m is the set of methods with which the constraint is associated, p is the the assertion that must be true, s is the set of subjects for which the constraint is enforced, e_{exp} is a set of events (messages) that will terminate enforcement of c, t_{start} is the time from which c is to be enforced, t_{exp} is the time at which c expires and a is the action that will be taken if p is not true. Stated differently, if any subject listed in s has sent any message μ listed in m, none of the events listed in e_{exp} has occurred and the time has not yet reached t_{exp} (but is not earlier than t_{start}) then p will be evaluated (after μ has been handled). If p evaluates to true, no further action is taken; if p evaluates to false the action specified by a is taken. In most cases in this paper the action specified will be to roll back the current sequence of messages, which will be indicated by $\hookleftarrow$. The only other action currently allowed is logging of the fact that the constraint has been violated. As a notational shorthand a $*$ will sometimes be used in the first position of the tuple to indicate that the constraint will be enforced for *any* message sent. Similarly, a $*$ in the third position indicates that the constraint applies to all subjects.

To illustrate, the constraint

PREVENT ACCOUNT:INCREASEDAILYFEE UNTIL PATIENT:RELEASE

is represented by the tuple

$\langle$ {Account:IncreaseDailyFee}, True, $*$, {Patient:Release}, 0, ∞, $\hookleftarrow\rangle$

The use of ∞ here indicates that no expiry time has been specified in the constraint. The smallest value for time is 0 and this has been used as the starting time of this constraint. Similarly, the constraint

VERIFY ACCOUNT.DAILYFEE = ACCOUNT'.DAILYFEE UNTIL PATIENT:RELEASE

is represented by the tuple

$\langle *$, Account.DailyFee = Account'.DailyFee, $*$, {Patient:Release}, 0, ∞, $\hookleftarrow\rangle$

The semantics of constraint checking can also be specified formally to remove the ambiguity associated with an informal description. Assume that a subject σ sends a message μ at a time τ. This message can be sent directly by the subject, or by a method activated by an earlier message of the subject. Also assume that the events that occurred prior to time τ are $e_{t_0}, e_{t_1}, e_{t_2}, \ldots, e_{t_n}$. Let C represent the set of all constraints which have been entered into the system at time τ. Now consider the pairs $\langle p, a\rangle$ in $\{\langle p, a\rangle | (c = \langle m, p, s, e_{exp}, t_{start}, t_{exp}, a\rangle \in C) \wedge (\mu \in m) \wedge (\sigma \in s) \wedge (\forall t)(t_{start} \leq t \leq \tau)(e_t \notin e_{exp}) \wedge (t_{start} \leq \tau \leq t_{exp})\}$. If p evaluates to true for any pair $\langle p, a\rangle$, the corresponding action a is initiated.

4.3 Enforcement

Enforcement of constraints is notoriously expensive (Sandhu and Jajodia, 1991; 1995), especially if implemented directly from notation such as that given above. However, we will argue that many worthwhile constraints can be enforced inexpensively.

Constraints can be enforced by dynamically modifying access controls (for PREVENT constraints) or by including special 'methods' in the object, to be automatically activated when necessary (for VERIFY constraints).

To handle VERIFY constraints we assume that access controls can be extended by adding methods to be executed when the object (or protected facet) is accessed. Such a

method can perform additional checks and abort execution if necessary. To support such methods is relatively simple—see, for example, Olivier (1995).

If such methods are supported, support of PREVENT constraints is also simple. Views may also be used to handle PREVENT constraints—see Hailpern and Ossher (1990) and Shilling and Sweeney (1989).

To enforce VERIFY constraints directly, it may be necessary to check all constraints after a user request has been handled. However, this is clearly prohibitively expensive. A first attempt to optimise this is to take the state of objects into account that may affect a particular constraint and then only verify that constraint if a method that has potentially modified such a state has been activated. The following algorithm will accomplish this:

> Let M be the set of methods referred to by the conditional portion of constraint c. Then, if constraint c was satisfied prior to some sequence of messages, c need only be verified if one of the methods in $C = f(M, M)$ has been activated during the sequence of messages, where f is defined as follows:
>
> *Function f(M, G)*
> *If* $M = \emptyset$ *return* $\emptyset$
> *Let V be the set of all instance variables read by methods in M*
> *Let R be the set of all methods (functions) that return a value used by methods in M*
> *Let W1 be the set of all methods that write a value to any variable in V*
> *Let W2 = f(R-G, G+R)*
> *Return W1* $\cup$ *W2*

A method that sets a flag associated with c can now be attached to every method in C. When a user request is about to terminate, only those constraints for which flags have been set during the handling of the request need to be verified. Stated differently, a constraint of the form $\langle *, p, *, e_{exp}, t_{start}, t_{exp}, a\rangle$ can be replaced by a constraint of the form $\langle C, p, *, e_{exp}, t_{start}, t_{exp}, a\rangle$, where C has been calculated as above.

Note that any of the sets used in the algorithm above may contain facets of more than one object. The algorithm works by identifying all methods that can change the state of objects on which the predicate directly depends (in $W1$) and those on which the constraint indirectly depends (in $W2$).

Further optimisation of constraint checking is possible: for a constraint of the form O.PRICE $\neq$ O'.PRICE the algorithm above allows the instance variable which contains the price (or those from which the price is calculated) to be written as long as the value returned by the PRICE method does not change. A simpler solution for this particular type of constraint is to interpret it that none of the related state is allowed to change—then a simple denial of write requests to any variable that appeared in V during execution of the algorithm above will implement the constraint without the necessity to check the constraint afterwards.

Note that even the first version of the algorithm may constrain the object more than intended since it may disallow modifications to variables that do not directly influence PRICE; for an improved implementation dataflow techniques, or even human intelligence, may have to be used. However, we assume that is is possible to attach methods to enforce constraints of the described form, that are not too expensive—even if more than what is absolutely required, is constrained.

5 FEDERATED DATABASES

As motivated earlier, constraints in a federated database have the additional purpose to ensure integrity in one database where integrity depends on the contents of another database—the expectations a site has about the behaviour and use of an object can be expressed using the same notation used in the previous section. However, enforcement is more problematic than in the standalone case: Here the information to be constrained is owned by (and probably occurs on) a different site from which it is accessed. We address this conflict by enabling the recipient to request guarantees from the service provider and the service provider to provide guarantees about the shared information. If the provided guarantees meet or exceed the requested guarantees, interoperation is possible. We consider comparing required and provided guarantees next. This is followed by remarks on the duration of guarantees. Finally, we consider implementation (enforcement) of constraints.

5.1 Requested versus provided guarantees

Guarantees are partially ordered and a user will use the service if the supplied guarantee satisfies (or exceeds) the requested guarantee. Unfortunately it is not always easy to compare such guarantees. To illustrate such comparisons, consider the following: It is easy to see that

PREVENT REFLETTER:SETTEXT UNTIL 1 DECEMBER 1998

exceeds

PREVENT REFLETTER:SETTEXT UNTIL 1 JANUARY 1998

However, without (probably human) knowledge about the semantics of the system

PREVENT ACCOUNT:INCREASEDAILYFEE UNTIL PATIENT:RELEASE

cannot be compared to

PREVENT ACCOUNT:INCREASEDAILYFEE UNTIL 1 JANUARY 1998

It is easy to see that a constraint $c_1 = \langle m_1, p_1, s_1, e_{exp,1}, t_{start,1}, t_{exp,1}, a_1 \rangle$ exceeds a constraint $c_2 = \langle m_2, p_2, s_2, e_{exp,2}, t_{start,2}, t_{exp,2}, a_2 \rangle$ if $m_2 \subseteq m_1$, p_2 is a factor of p_1, $s_2 \subseteq s_1$, $e_{exp,1} \subseteq e_{exp,2}$, $t_{start,1} \leq t_{start,2}$, $t_{exp,1} \geq t_{exp,2}$ and $a_1 \geq a_2$. (For actions rollback exceeds logging.)

Not only is it not easy to compare every pair of comparable restrictions, it is also not easy to verify all restrictions—see the next section. Fortunately, it seems that most realistic restrictions are fairly simple to specify (and hence to compare to others) and also to enforce. Further, since the provided guarantee is allowed to exceed the requested guarantee, the service provider may be willing to provide a guarantee that may guarantee more than required, but which is easy to verify. To illustrate, consider the following example: A supplier may share a price list with clients. While the clients may only request that prices do not increase for a given time after a price has been obtained, the supplier may prefer to guarantee that prices will not be modified at all for this period (or a longer period). It is relatively simple to verify that an object has not been modified at all, compared to verifying that it has only been modified in some given way.

Since different users may (realistically) expect different guarantees from a service pro-

vider, the service provider may give a guarantee that forms some common upper bound of such expected guarantees.

5.2 Guarantee duration

Obviously guarantees cannot always be given indefinitely: a guarantee that a price will not be increased after a quotation does not mean that the price will never be increased again. A guarantee that a diet for a patient will at least be available does not mean that it will always be available. Approval of an account by a medical aid does not make any further sense after the account has been paid by the medical aid.

A guarantee can be given for a specific time period—three hours, seven days, or whatever time period may be adequate. Alternatively, a guarantee can be given until some event occurs: at the time the account is paid by the medical aid, the guarantees given by the specialist are no longer required. Another interesting option is to use versioning: A general reference may be given to be used by clients to obtain prices. However, once a client uses the service to obtain the price, the client can be given a reference to a specific version to use as basis for the given quotation. This version is guaranteed not to be updated for some period, but the general service (for other clients) can still be updated by the supplier. Note that versioning will, in general, be used in conjunction with an expiration time (or expiration condition).

5.3 Implementation

Enforcement of guarantees in a single site environment has already received attention in section 4.3. However, the federated environment poses an additional problem: enforcement of a guarantee obtained from an essentially independent other site.

There are a number of approaches to assure conformance with provided guarantees: They can either be enforced in realtime, or only verified when used and, if contravened, appropriate steps can be taken. Enforcing guarantees is more complex than only verifying guarantees, but, obviously, preferred over verification. The option chosen partly depends on the trust relationship between a service provider and the recipient.

Guarantees can be enforced in a number of ways: The concerned object can be relocated to a site trusted by both service provider and recipient, who will then enforce the guarantees; the software supporting the federation can be written such that the recipient is ensured that the guarantees will be enforced, wherever the concerned object may be (even at the service provider's site) or the object can be relocated to the recipient who can then self enforce the guarantees. Enforcement by the designated site is then performed in essentially the same way as in the standalone case discussed in section 4.3.

If the federation is designed to enforce constraints, the following approach may be used. We do not consider alternatives in detail. Firstly, mutual trust between sites can be established by implementing the database according to the SPO model (Olivier, 1995). The protection associated with an object will then be enforced by all members of the federation, irrespective from which site the object is accessed, or to which site the object is relocated. SPO also protects entities with the aid of methods that are executed when a subject attempts to access the entity; this method performs the required access check and aborts execution if necessary—this facility will also be used to enforce integrity constraints.

Secondly, the object has to be analysed so that the methods that flag the necessity

to verify a constraint (see section 4.3) can be associated with the appropriate methods. This 'analyser' can be incorporated into all sites, or a special analysis site (or sites) can exist in the federation. Such analysis cannot be done before the object is entered into the system (unless the compiler or similar software at all sites can be trusted). If an object is represented internally at a relatively high level, analysis of an object can be performed when the object has already been entered into the trusted portion of the database. In the case of the SPO prototype (Olivier, 1996), objects are represented as tables of data accompanied by methods in postfix notation. The algorithm given in section 4.3 can easily be applied to code represented in this notation. Analysis may require more than one object to be relocated to the analysis site. The analysis site can record the results of the analysis for the constraint—it does not have to be repeated for other objects of the same class.

After analysis has been completed the 'constraint flagging' methods can be added to the methods identified by the analysis. Logically this can be done by yet another site: the certification site. (Physically this may be the same site used for analysis and/or other purposes.) Since the SPO model guarantees that the protection added by the certification site will be enforced at any site, the concerned object(s) can safely be relocated back to another site in the federation. The certification site records that the guarantee has been implemented.

When a user enters a message into the system it can now be handled in the usual way. However, when a potential constraint violation occurs, the 'constraint flagging' method will notify a trusted component at the sending site to verify the particular constraint. (This trusted component is a new component of the TCC in SPO.) When handling of the user message is complete, or when any information is to be given to the user, all such 'flagged' constraints are verified and execution rolled back if necessary.

During operation, guarantees can be supplied by the service provider before requested and/or the supplier can be requested to supply a guarantee. If a subject now wishes to use a service, but expects acceptable guarantees before using the service, this subject can determine (directly or indirectly) from the certification site whether the guarantee has been given and, if not, request it. If an acceptable guarantee has not yet been given, the owning site will be requested to relocate the object to a certification site (perhaps via an analysis site) with the guarantee it is willing to provide. The certification site then informs the subject about the existence of the guarantee. Hereafter the subject can access the service normally for as long as it considers the certified guarantee adequate. When the guarantee is no longer adequate, the certification process can be repeated.

If realtime enforcement of guarantees is not considered necessary, guarantees can be verified. For this a number of options also exist: Firstly, a site may be required to sign any values returned by it that are bound by a guarantee (or when requested); such values can also be requested via a trusted third party and logged by that party. In all cases here the guarantee will also have to be available in a form that can serve as proof that it had been given: When issued it can be signed electronically or be sent to a trusted third party. We do not consider details in this paper.

Obviously, one possibility for a site who no longer wants to be bound by a guarantee (and where the object has not been relocated to a trusted third party) is to simply make the object unavailable until the guarantee expires. There is therefore also a need for availability guarantees (or an automatic extension if the shared information is not available when an access attempt is made before a guarantee expires). We do not discuss this problem further in the current paper, but note that some (federally accepted) party

has to exist in the system who can certify that a service was indeed not available at some point if a recipient claims it. Steps that can then be taken fall outside the scope of this paper.

6 CONCLUSION

This paper considered the use of integrity constraints in federated databases. Without support for such constraints (or guarantees) interoperation of databases will be restricted. A system that automates workflow and involves more than one organisation, for example, will be much simpler to implement if guarantees are supported. Without such guarantees, each organisation will be responsible for verifying that the other party has acted according to an agreement. If this agreement has been violated a complex series of actions will ensue—too complex to automate and also too lengthy for many time-critical processes or transactions. Further, guarantees simplify interoperation between parties who share a federated database: If both parties accept the use of guarantees, no new contract or other agreement may be required for new forms of business between such parties. If guarantees already in place are acceptable, transactions can start immediately.

The notation used in this paper emphasised readability rather than ease of enforcement. Additionally, enforcement of constraints is known to be costly. However, it has been argued that enforcement of typical guarantees in the described environment is realistic. More work, considering additional examples, has to be done to confirm this claim.

Domain integrity is another aspect of integrity that influences interoperation between sites. In an object-oriented environment such integrity can be enforced by the class mechanism (and, if required, code encapsulated in the objects). However, in a federated database changes to the definitions of classes can adversely affect users of the service and therefore classes also have to be subject to guarantees. This type of guarantee needs further research attention.

REFERENCES

Abrams, M.D., Amoroso, E.G., LaPadula, L.J., Lunt, T.F. and Williams, J.G. (1993) Report of an Integrity Research Study Group. *Computers & Security*, **12**, 679–89.

Bertino, E., Bettini, C., Ferrari, E. and Samarati, P. (1996) A Decentralized Temporal Authorization Model, in *Information Systems Security—Facing the Information Society of the 21st Century* (eds. S.K. Katsikas and D Gritzalis) Chapman & Hall, London, 271–80.

Ceri, S. and Widom, J. (1993) Managing Semantic Heterogeneity with Production Rules and Persistent Queues. Proceedings of the 19th VLDB Conference, Dublin, Ireland, 108–19.

Chawathe, S.S., Garcia-Molina, H. and Widom, J. (1993) *Constraint Management in Loosely Coupled Distributed Databases.* Technical report, Computer Science Department, Stanford University.

Chawathe, S.S., Garcia-Molina, H. and Widom, J. (1996) A Toolkit for Constraint Management in Heterogeneous Information Systems, in *Proceedings of the Twelfth Interna-*

tional Conference on Data Engineering (ed. S.Y.W. Su) IEEE Computer Society, Los Alamitos, CA, 56–65.

Date, C.J. (1983) *An Introduction to Database Systems, Volume II.* Addison-Wesley, Reading, Massachusetts.

Dayal, U., Hanson, E. and Widom, J. (1995) Active Database Systems, in *Modern Database Systems: The Object Model, Interoperability and Beyond* (ed. W. Kim) ACM Press, New York, New York, 434–56.

Eswaran, K.P., Gray, J.N., Lorie, R.A. and Traiger, I.L. (1976) The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, **19**, 11, 624–33.

Fernandez, E.B., Summers, R.C. and Wood, C. (1981) *Database Security and Integrity.* Addison-Wesley, Reading, Massachusetts.

Grefen, P. and Widom, J. (1996) Integrity Constraint Checking for Federated Databases. Proceedings of the First IFCIS International Conference on Cooperative Information Systems, Brussels, Belgium, 38–47.

Hailpern, B. and Ossher, H. (1990) Extending Objects to Support Multiple Interfaces and Access Control. *IEEE transactions on Software Engineering*, **16**, 11, 1247–57.

Kappe, F. and Maurer, H. (1994) From Hypertext to Active Communication/Information Systems. *Journal of Microcomputer Applications*, **17**, 333–44.

Kim, W., ed. (1995) *Modern Database Systems: The Object Model, Interoperability and Beyond.* ACM Press, New York, New York.

Lunt, T.F. (1995) Authorization in Object-Oriented Databases, in *Modern Database Systems: The Object Model, Interoperability and Beyond* (ed. W. Kim) ACM Press, New York, New York, 130–45.

Olivier, M.S. and Von Solms, S.H. (1994) A Taxonomy for Secure Object-oriented Databases. *ACM Transactions on Database Systems*, **19**, 1, 3–46.

Olivier, M.S. (1995) Self-protecting Objects in a Secure Federated Database. Ninth IFIP WG 11.3 Conference on Database Security, Rensselaerville, New York.

Olivier, M.S. (1996) Self-protecting Objects in Multipolicy Federated Databases: A Prototype. *In preparation.*

Özsu, M.T. and Valduriez, P. (1991) *Principles of Distributed Database Systems.* Prentice-Hall, Englewood Cliffs, New Jersey.

Rabitti, F., Bertino, E., Kim, W. and Woelk, D. (1991) A Model of Authorization for Next-Generation Database Systems. *ACM Transactions on Database Systems*, **16**, 1, 88–131.

Rusinkiewicz, M., Sheth, A. and Karabatis, G. (1991) Specifying Interdatabase Dependencies in a Multidatabase Environment. *IEEE Computer*, **24**, 12, 46–53.

Sandhu, R. and Jajodia, S. (1991) Integrity Principles and Mechanisms in Database Management Systems. *Computers & Security*, **10**, 413–27.

Sandhu, R. and Jajodia, S. (1995) Integrity Mechanisms is Database Management Systems, in *Information Security: An Integrated Collection of Essays*, (eds. M.D. Abrams, S. Jajodia and H.J. Podell). IEEE Computer Society, Washington, DC.

Sheth, A.P. and Larson, J.A. (1990) Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, **22**, 3, 183–236.

Shilling, J.J. and Sweeney, P.F. (1989) Three Steps to Views: Extending the Object-Oriented Paradigm. *Proceedings of the Conference on Object-Oriented Programming*

Systems, Languages and Applications, ACM, 353–61.
Thuraisingham, B. (1994) Security issues for federated database systems. *Computers & Security*, **13**, 509–25.
Woelk, D., Bohrer, B., Jacobs, N., Ong, K., Tomlinson, C. and Unnikrishnan, C. (1995) Carnot and InfoSleuth: Database Technology and the World Wide Web. *Sigmod Record*, **24**, 2, 443–44.