# Semantic Integrity for Persistent Objects

Peter J Barclay and Jessie B Kennedy
Napier Polytechnic, Craiglockhart Campus
219 Colinton Road, Edinburgh EH14 1DJ
e-mail:
pete@uk.ac.napier.cs and jessie@uk.ac.napier.cs

**Abstract**

Modelling constructs for specifying semantic integrity are reviewed, and their implicit execution semantics discussed. An integrity maintenance model based on these constructs is presented. An implementation of this model in a persistent programming language is described, allowing flexible automated dynamic integrity management for applications updating a persistent store; this implementation is based on an event-driven architecture.

persistent programming, conceptual modelling, semantic integrity, active object-oriented databases, code generation

## 1 Introduction

Napier88 [MBCD89], [DCBM89] is a high-level, strongly-typed, block structured programming language with orthogonal persistence [Coc82]; that is, objects of any type created by programs can outlive the execution of the program which created them. Persistent objects can be reused in a type-secure way by subsequent executions of the same program, or by other programs.

Persistent languages are well-suited to the construction of data-intensive applications [Coo90]; programs are written to manipulate data, and the inbuilt (and transparent) persistence mechanism provides for its storage and retrieval.

This article describes an integrity management system (IMS) written in Napier88; this forms part of a larger system which supports the development of persistent application systems [BK92]. This integrity management system allows its user to specify constraints on data in a high level, declarative notation, and then ensures that the data respects these constraints. The

system infers what events could compromise the integrity of the data, and then on these events checks those objects which could have been affected. Further, the system provides *activeness* for the database[1], by allowing the specification of Condition-Action rules, called *triggers*, which call procedures automatically whenever specified conditions are met. The operation of this integrity management system is transparent to application programmers.

Section 2 overviews the type of integrity information which can be specified for this system, with comparisons to those offered by some recent database systems; section 3 examines constructs, provided by the system, to allow management of *how* integrity is maintained; section 4 overviews the implementation of the system.

## 2 Specifying Integrity

Specifying integrity constraints is part of the process of information modelling. Attaching a collection of constraints to some data refines the precision with which that data is described, and may lead to a greater understanding of the data. Specifying constraints is also part of the process of database design; any integrity constraints which can be supported by the database reduce the task of the application programmer (since she need not code these constraints), and increases confidence in the integrity for the data (since this is under centralised control). However, the level of support for integrity provided by many database systems is not high, although it has been estimated that as much as 80% of a typical database definition may be concerned with integrity specification [Dat87, page 455].

In succeeding sections, some mechanisms available for expressing constraints on data will be considered. The notation used for examples is NOODL (Napier Object Oriented Data Language), a conceptual-level object oriented data description language based on the data description notation used in [BK91]. This language is fairly representative of various recent object oriented data description languages, but has the advantage of not being tied to any particular database management system (DBMS); it will be used to discuss integrity specification in general, and will also serve as source code to specify integrity information to the IMS.

The term 'constraints' here is intended to mean *explicit* constraints that capture some additional fact about the real world enterprise being mod-

[1]In this article, the term *database* is used, rather loosely, to mean a collection of data in Napier88's persistent store which is described by some particular application schema.

```
class Employee
ISA Person
property
  wage: Money ;;
constraint
  employable_age is
    self.age >= 16 and self.age <= 65 ;;
```

Figure 1: Employee Schema (A)

```
class Circle
properties
  radius: Number ;;
  area: Number ;;
constraint
  area_rule is
    self.area = pi * self.radius * self.radius ;;
```

Figure 2: Circle Schema (A)

elled, rather than constraints implicit in the data model chosen [TL82]. Note that NOODL incorporates integrity specification with inheritance, since constraints (and, as described later, triggers) on any class are inherited by its subclasses, where they may optionally be overridden (redefined).

## 2.1 Predicate Based Constraints

A predicate based constraint simply says that some fact is true of the data. An example is shown in the NOODL schema in figure 1 where it is asserted that an employee must be between 16 and 65 years of age, or in figure 2, where it is asserted that the area of a circle must be $\pi$ times its radius squared.

The normal method of enforcing such a constraint is to forbid updates which violate it; for example, the DBMS would refuse to allow an application program to update an employee's age to 5. This is the principle behind the *define integrity* construct of Ingres [Dat87], the *data restriction* of Generis

```
class Circle
properties
  radius: Number ;;
  area: Number is
    pi * self.radius * self.radius ;;
```

Figure 3: Revised Circle Schema (B)

[gen90, pages 4/12 - 4/19], and *ic* command of FDL [Pou88].

However, other options are possible. Constraint-satisfaction techniques [Lel88] allow relationships to be specified among a number of objects, and then, when provided with values for some of these objects, can find values for the others which maintain the constraint. In particular application areas, constraint satisfaction has proved very effective [Bor77]. Using such techniques one could envisage a database which would allow a user to reset either the radius or the area of a circle object, and would reset the other value in accordance with the constraint labelled `area_rule`. However, constraint-satisfaction systems are difficult to implement, do not run fast, and are application-specific, typically handling numeric constraints; therefore this approach has not generally been used for enforcing constraints in conventional database systems. Further, the user must be prepared to have *unknown* as a value for some properties.

## 2.2 Derived Properties

Many newer database systems also offer *derived properties*, which provide an alternative way of expressing some constraints implicitly. Examples of these are the *tuple functions* of Postgres [pos90], or the *derived functions* of Iris [LW91].

As an example, the circle schema in figure 2 could be rewritten as in figure 3.

The first schema (A) has a symmetry of expression absent in the second. It states that a circle will have a radius and an area, and that the relationship between these two quantities is as expressed in the constraint labelled `area_rule`. The second schema (B) removes this symmetry in expression of the constraint by showing how the area may be derived from the radius. The implicit execution semantics are, that in case (A) the user may update

either property, but in (B) only the radius is updatable (which requires a little extra effort if it is the new area that is known).

These implicit execution semantics suggest that representation (A) is more conceptually accurate in this case. However, it may be required to model situations where one property genuinely is conceptually derived from others. For example, the profit of a company may be found by adding together its various sources of income, and deducting taxes, payments to employees and other outgoings. It is unlikely that a user would wish to assign arbitrarily a new value to the profit rather than to one of the contributing factors.

Note that the expression showing how the property is to be derived is a 'conceptual' specification of its derivation; it is not an indication of how values are actually stored or computed in some implementation, since optimisations may be applied.

## 2.3 (Event)-Condition-Action Rules

Another mechanism provided to support integrity is the *rule*. Deductive databases have extended this to support sophisticated inferencing, whereas other systems support simple rules only. For example, HiPAC [Day88] supports Event-Condition-Action rules which represent asynchronous actions associated with a change of state. These are similar to the *self-triggering* rules of OZ+ [WL89]. Postgres supports rules [SJGP87] the basic format of which is

```
ON event TO object
WHERE condition
DO action
```

NOODL provides Condition-Action rules, introduced by the NOODL reserved word *trigger*, which specify the action to occur when some condition is met. It is also possible to obtain the functionality of If-When rules[2] and Event-Action rules[3] using this construct.

We may consider the constraints described earlier to be a special case of these Condition-Action rules, where the action to be taken is the abortion

[2] 'if A then B' is equivalent to 'not A or B'.

[3] 'on update to property do action' can be expressed as 'property = zerovalue or true : action', since the predicate will be evaluated on any update to the property, and will always be true. A suitable syntactic sugaring for this may be provided in the future.

```
class Employee
ISA Person
property
  wage: Money ;;
operation
  retire is ... ;;
constraint
  min_age is
    self.age <= 16 ;;
trigger
  retiral is
    self.age >= 65 : self.retire ;;
```

Figure 4: Revised Employee Schema (B)

of an offending process. The word *rule* will therefore be used as a generic term for constraints and triggers together. However, the two concepts are distinguished for a number of reasons. Firstly, a trigger fires when the predicate describing its condition-part is true, whereas a constraint aborts a process when its predicate is false. It will be seen later that it is also useful to be able to manage the enforcement of the two constructs separately; for example, suspended triggers are permitted to persist unfired when a program terminates, but suspended constraints may not remain unverified.

Rules permit the modeller to capture more semantics about data, particularly about its behaviour. For example, the schema in figure 1 can be revised to that in figure 4. Here, the trigger `retiral` introduces a rule which says that if an employee becomes 65 or older, the operation `retire` (not defined here) should be applied to her. (Some suitable mechanism can be used to prevent re-firing of the trigger `retiral` on further updates of the age, preferably by migrating the object to a new class Pensioner for which no such trigger is defined).

However, rules introduce procedurality into the specification; it can be hard to foresee the consequences of a large number of (perhaps mutually activating) rules being fired, and such a system is not necessarily deterministic. One possible solution is to introduce rule *priorities*, as described in [ACL91]. (The system described in section 4 does not support priorities, but is deterministic in the sense that the same transaction, run on the same

database state, will always produce the same sequence of rule-firing).

## 2.4 How a Constraint is Enforced is Part of its Meaning

The database designer has almost too much choice for how to specify her understanding of the data; but the choice of expression carries some extra information. For example, *value propagation* [BCG+87] may be represented by any of the above mechanisms. If it is required to specify that the door of a car must be the same colour as the body, then an explicit predicate based constraint assumes that both colours should be updated together. With a derived property, the colour of the door can be made subordinate to the colour of the body. Using triggered updates, subtleties can be represented such as saying that changing the colour of the car changes the colour of the door, but not *vice versa*; a red car has a red door, but if the door is expressly made blue, that is a customisation and nothing is to be assumed about the colour of the body.

Ideally, there should be a larger number of integrity specifications, each embodying one fact about the model, rather than fewer, each expressing more information. This not only simplifies reading the schema and enables the database to report violations more meaningfully, but discourages oversights like failing to specify that the reason an employee is not aged over 65 is that she will have retired.

In the above example, the integrity specification has progressed from the assertion that an employee is aged between 16 and 65 to the assertions that an employee is aged over 16, and that an employee becoming older than 65 retires. Consider two programs updating the database such that one sets the age of an employee to be under 16, and the other sets the age of an employee to be over 65. In the first case, it is known only that the updating program is in error[4]; but in the second case it was known in advance that the reason an employee could not be older than 65 is that she would have already retired; hence, by simply forbidding this by a predicate, as in figure 1, one fails to capture part of the meaning of the constraint.

Thus how a constraint is enforced is part of its meaning, and it is desirable for a data model to provide a variety of methods of constraint enforcement. Ideally, the modeller may understand in advance what a violation would signify, and install a rule to readjust the database accordingly. A constraint may be specified implicitly by showing how the value of one property

[4] Treatment of *exceptional values* in data is outwith the scope of this article.

is derived from others, so that a query on the derived property will produce the correct value. Finally, a predicate may be specified which data must satisfy, precluding updates which violate it. Here the modeller is admitting that she cannot foresee the circumstances of a violation (of course, this is sometimes unavoidable). Unfortunately, it is not necessarily the case that it is the update which violates the constraint, rather than some earlier update, which is in error, especially where the constraint involves a large number of objects; but this may be resolved since a failed update will usually entail the intervention of a human.

A predicate-based constraint offers the greatest declarativeness of specification, but in a conventional DBMS the method of maintaining the constraint is typically simply to abort any transaction which violates it. Rules allow more capture of behaviour of the data, at the cost of the introduction of some procedurality; Event-Condition-Action rules require the user to state after what events activation of the rule is to be considered. Derived properties provide a good mechanism to express the constraints implicit in 'emergent' and 'immutable' properties (where respectively the expression to be evaluated is or is not a function of the state of other database objects); in some contexts however they may introduce an artificial asymmetry into the specification.

# 3 Managing Integrity

This section considers the model according to which semantic integrity is maintained. The kinds of rules the programmer may wish to specify over her data have been reviewed; without yet considering how to implement these rules, it is necessary to provide mechanisms for the management of their enforcement.

The traditional model of integrity enforcement in database systems is the transaction model (see, for example, [EN89, chapter 19]); the transaction is a construct which combines atomicity, serialisability, and recoverability. In [Sut90], Sutton argues that many systems require a more flexible approach to the maintenance of consistency. Rather than have all-or-nothing consistency, specific processes may require the enforcement or violation of specific constraints, regardless of whether they are enforced generally; moreover, specific processes may wish to choose *when* and *where* integrity is guaranteed, or may be violated.

It is desirable for the integrity maintenance model adopted to be as gen-

eral as possible within certain restrictions. The principle restrictions are as follows. Presently, Napier88 is a single-user system; there is no possibility of concurrent processes attempting simultaneously to access the store, so concurrency management constructs are not required. Another restriction arises through the way the persistent store works. As objects are referenced in a program, they are mapped from disk into memory, where they may be updated. When a `stabilise` operation is performed, updated memory objects are mapped back onto disk. Therefore, in order to prevent updates from being committed when a transaction fails, the mapping back to disk of memory objects must be prevented, which unfortunately requires the abortion of the process to which the memory is allocated. This means that there is no possibility of *persistent transactions*[5], or of *nested transactions* [Mos81], since uncommitted local copies of objects cannot survive a `stabilise` operation (or, equivalently, program termination).

(This restriction depends on the fact that it is natural, but not ideal, to build the `commit` operation, which controls the conceptual consistency of the data, on the `stabilise` operation, which manages the persistence of program objects. In a system where uncommitted local copies of objects could persist, this restriction would be lifted. Discussion of such a system is outwith the scope of this article).

### 3.1 Integrity Management Constructs

The constructs described below are provided by the IMS; the argument to any of the first five constructs is a list of the constraints to which it applies, or one of the shorthand tags `TRIGGERS`, which applies it to all triggers, `CONSTRAINTS`, which applies it to all constraints, or `ALL` which applies it to all of both. The `commit` construct requires no arguments.

**enforce** ensures that any subsequent operations respect the constraints specified; if a violation occurs, the violating process is aborted. Specified triggers are fired when the appropriate condition is met.

**ignore** allows subsequent operations to violate the specified constraints freely; specified triggers are never fired when their activation conditions are met.

[5]Persistent transactions would reintroduce concurrency management issues even in a single-user system, since different transactions, run in stages by sequences of programs, could be interleaved.

**suspend** also allows subsequent operations to violate the specified constraints; however, any updated objects which might be in violation of some constraint are logged. Similarly, triggers whose activation conditions are met are logged (but not fired).

**status** simply shows whether the constraint or trigger is enforced, ignored or suspended.

**clear** clears the logs created by `suspend`. Any logged triggers are fired, and any logged constraints checked; failure of a constraint check aborts the process.

**commit** commits all updates since the last commit operation. Since persistent transactions are forbidden, an attempt to commit while logged objects are still unverified will abort the process attempting to commit.

It should be noted that a constraint is enforced, suspended or ignored for the entire class of objects on which it is defined. The difficulties of attempting efficiently to apply rules to individual objects is described in [SRH].

By including these constructs in her programs, a programmer may have detailed control over how integrity is maintained. However, it is her own responsibility to ensure that the constructs used interact as intended. For example, the traditional transaction model can be extended to include procedure calls, rather than simple reads and writes, within a transaction. Now, if the programmer intends to run an *assertion-transaction*, she might place `enforce(THIS_CONSTRAINT)` at the beginning of the block which constitutes the intended transaction; it must be ensured that no procedure called from within this block contains an uncancelled `suspend(ALL)` construct, which would destroy the semantics of the intended transaction.

Whereas this problem can be avoided by ensuring that every procedure leaves its integrity maintenance context unaltered, a better approach is perhaps to use the integrity management constructs to build transaction constructs of the required type, and use these except where occasion demands more detailed control. Within the above-mentioned restrictions of the persistent programming approach, the constructs provided should be sufficient to build any required transaction primitives such as those for conventional flat transactions, assertion-transactions and repair-enforce transactions (see [Sut90] for more details). Flat transactions are considered below as an example.

Figure 5 shows how a conventional flat transaction, providing integrity, recoverability and atomicity, might be constructed. The figure defines a

procedure, `transact`, which runs another procedure, `updates`, in a context where all constraints are suspended. If the updates are successful, they are committed, otherwise the transaction is aborted; the transaction records its progress in a transcript file as it proceeds. For convenience, it is assumed that the procedures `save_status` and `restore_status` (built from the `status` construct) have already been defined, so that the transaction can leave its its integrity maintenance context unaltered. If the user has a procedure `my_updates` which alters the persistent store, she may run it as a transaction by calling `transact(my_updates())`. Atomicity is provided by running the updates inside a procedure call.

When used with constraints, the integrity management constructs can provide a very flexible form of transaction functionality. However, these constructs are also useful in controlling the enforcement of triggers. For example, triggers may be activated and deactivated using `enforce` and `ignore`; in this way, once-only triggers (*eg*, [Hug91]) may be implemented.

Note that `commit` will succeed (with appropriate warning) while logged triggers (but not constraints) remain unfired. In this way, trigger execution may be:

**immediate,** using `enforce(TRIGGERS)`;

**delayed,** using `suspend(TRIGGERS)` and `clear(TRIGGERS)`; or

**detached** (run in a separate transaction), using `suspend(TRIGGERS)`, `commit`, and `clear(TRIGGERS)`.

Clearly, the semantics of an updating program will depend on which coupling mode is used. (Consider the effects of an *annual-review* transaction, which triggers a 5% pay increase for all employees, followed by a transaction which fires all employees earning over £20000).

## 4 Implementing Integrity

The kinds of statements one might like to make regarding the integrity of data, and how control of this integrity might be managed in programs, have been discussed; now it remains to describe how support for this integrity maintenance is to be implemented. The ideal is to allow a programmer to specify in a conceptual notation what constraints are required, and then allow her to write programs manipulating this data, secure in the knowledge that the integrity specified is being maintained. No code should appear in

```
! construct to support conventional transaction
! ----------------------------------------------

let transact = proc(updates: proc())

begin ! transact
let t_id = id_giver() ! assign ID to transaction
transcript("transaction " ++ t_id ++ " initiated'n")
if transacting do {
  transcript("nested execution of transaction " ++ t_id ++ " attempted'n")
  abort()             ! Napier88 predefined abort procedure
  } ! if
commit()
transcript("transaction " ++ t_id ++ " precommitted'n")
save_status()
suspend(CONSTRAINTS)
transacting := true   ! set global flag to prevent nesting
updates()             ! run the user's update procedure
transacting := false  ! unset flag
restore_status()      ! restore integrity maintenance context
transcript("transaction " ++ t_id ++ " ran updates'n")
clear(CONSTRAINTS)    ! check suspended constraints, fire suspended triggers
commit()              ! everything is ok if it got this far, so commit changes
transcript("transaction " ++ t_id ++ " postcommitted and terminated'n")
end ! transact
```

Figure 5: Construct for Conventional Flat Transaction

user programs to verify constraints or fire triggers; this would not only complicate application programming, but also reduce confidence in the integrity of the database. Evolution of the application schema is also considerably complicated when the code which maintains integrity is scattered among user programs. Moreover, it is required that constraints are verified and triggers fired whenever necessary, but there should be minimal redundant checking. Given that a constraint may be considered to be a special case of a trigger, the mechanism will be described with respect to constraints without loss of generality.

In the first subsection an event-driven integrity management system is described, providing automated support for specified integrity; the second subsection describes how the construction of such an integrity management system itself can be automated.

## 4.1 An Event-Driven Semantic Integrity Management System

In order for data in the persistent store to respect the integrity constraints defined on it, it is required that the necessary constraints be checked whenever an event occurs which might cause the constraint to be violated. These checks are to be transparent to the user, providing an *active* interface to the store for application programs. Adding such activeness to database systems is an area of current research; it is novel in a language with orthogonal persistence.

Event-driven architectures are useful where systems must respond to unpredictable conditions, or facilitate reconfiguration (*eg*, [SC91]). In order not to penalise data access unduly, the IMS requires an architecture where there is no busy-waiting and no examination of irrelevant events.

The event-driven architecture which supports maintenance of integrity was inspired by, but differs substantially from, the one described in [CK87] which forms the basis of the windowing system WIN (Windows In Napier88) [CDKM89]. Since architectures used to support windowing systems are more familiar, the two architectures are contrasted in figure 6.

In the first architecture (left), a procedure called an *event monitor* continually polls for (keyboard or mouse) events; these are passed to a *notifier*. Applications register with the notifier, passing it a boolean-valued procedure which determines whether they are interested in a particular event. The notifier passes the event down the list until it finds an interested application, to which it passes the event. This application may be another notifier, so

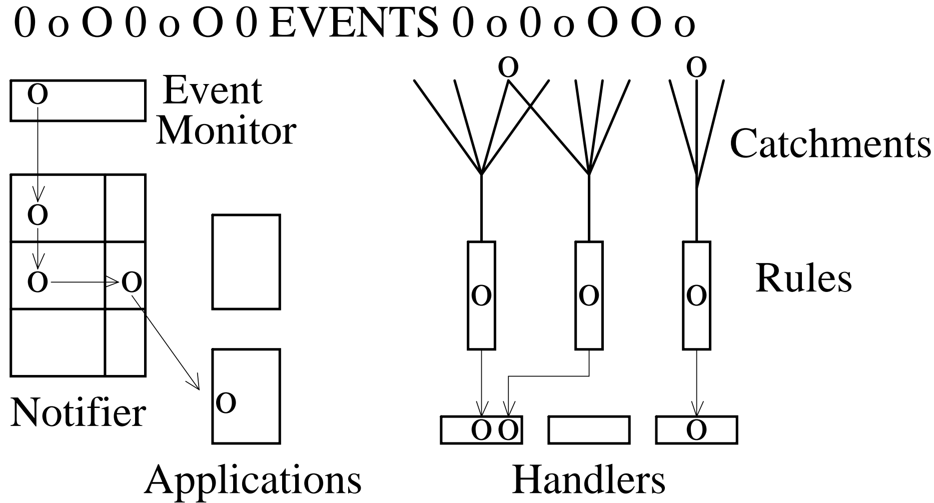


Figure 6: Event-Driven Architectures

a hierarchically nested architecture may be built up. An event in which no application is interested is discarded. The notifier hierarchy may be reconfigured dynamically, to represent changes in the layering of windows on the screen and so on.

The second architecture uses a structure called a *catchment*; this is a collection of procedures which monitors the set of all events which may lead to violation of a given constraint. Each constraint or trigger is implemented by a *rule object*. (A rule object is an object within the system, but not within the database with which the user interacts; treating rules as objects is discussed in [DPG91]).

Each rule has its own catchment, and receives any event which occurs within it. Unlike in the first architecture, the receiver of an event is not determined dynamically; it is determined statically which events may violate a constraint, and an appropriate catchment installed to capture them; in this way, the operation of the IMS does not unduly slow data access by processing irrelevant events. In a windowing system, user input is usually intended for some particular window; hence only one application receives an event. However, in the IMS, an event may potentially violate several constraints, and so fall into several catchments; hence it must be distributed to **all** the appropriate constraints for checking. In the catchment is a procedure which determines which object may have had its integrity violated by the event

```
class Employee
ISA Person
properties
  wage: Money ;;
  dept: Department
        \ staff ;;
constraints
  min_age is
    self.age >= 16 ;;
  pay_policy is
     self.wage < self.dept.head.wage ;;
```

```
class Department
properties
staff: setof Employee
     \ dept ;;
head: Manager
    \ manages ;;
```

```
class Manager
ISA Employee
properties
  manages: Department
         \ head ;;
```

Figure 7: Example Schema

(not necessarily the object to which the event occurred!); this object is passed to the *handler*.

A handler is an object which determines what is to be done with the object passed down from the catchment, in the context of a particular rule; it may for example check a condition, call a triggered procedure, log an object, or abort the current process. Each constraint is registered with one of several possible handlers. The integrity management constructs of section 2 largely work by registering rules with different handlers.

### 4.2 Example

It remains to be described what actually constitutes an event, and how it is captured. For clarity, rather than examine the algorithms used by the IMS, we shall consider as an example the maintenance of the constraints in the schema shown in figure 7. A particular instance of an employee object, and its associated department and manager objects, is shown in figure 8; the database will contain many such instances.

The constraint `min_age` applies only to the employee instance itself; the only property cited by the predicate which is the body of the constraint is *age*; hence the only event which can violate the constraint is an update to the employee's age. To maintain the constraint, it is necessary to trap all

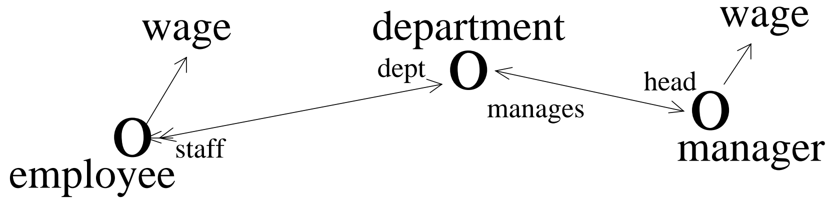


Figure 8: Example Database Subgraph

such updates, and check the integrity of the employee instance updated.

The constraint `pay_policy` is a constraint over the configuration[6] of the employee, department, and manager instances, and the wage properties of the first and last. Changes to the wage properties, or to the configuration, could violate the constraint. Hence, if the employee instance has its wage property updated, the integrity of this instance must be checked; and, if a manager instance has its wage updated, it is necessary to check the integrity, not of this manager instance, but of the employee instance to which it is related through its department[7]. Integrity must also be checked if an employee is assigned to a new department, or a department is assigned a new head; in the first case, the employee instance itself must be checked, in the second case, the employees which are the value of the staff property of the department. The obverse[8] updates, of assigning new staff to a department, or assigning a new department to be managed by a manager, also require a check to be performed. These events, and the objects which must be checked, are summarised in table 1; *self* in the second column refers to the object updated.

Determination of which object(s) to check after a given update relies on being able to track back along obverse links to the object(s) on which the constraint is defined; hence it is assumed that these obverses are available. However, this is not the case where a link is through a derived object-valued property, the derivation of which involves the result of some query over the database. The reason is, that in general it is not clear that the query can be inverted in order to find the object whose integrity may have been

[6]Rules over the components of complex objects may be treated similarly.

[7]In fact, one must check the integrity of each instance of the set-valued property staff of the department instance attached to this manager instance. For clarity, the fact that the integrity check must be mapped over this set is ignored here.

[8]By *obverse*, we mean the intuitive inverse of a potentially set-valued property (see [BK91] for details); the obverse of a property follows a backslash in a NOODL schema.

| Event | Object(s) to Check |
|---|---|
| update wage of employee | self |
| update wage of manager | self.manages.staff |
| update department of employee | self |
| update staff of department | self.staff |
| update head of department | self.staff |
| update ‘manages’ of manager | self.manages.staff |

Table 1: Events and affected Objects

violated. This defeats the efficient implementation of a check, since if it is not known which objects may have been affected, the whole database must be checked. For this reason, the prototype system does not allow the definition of constraints over object-valued derived properties, the derivation of which includes a database query.

It will be clear that the events of interest are pairs, consisting of a property-update, and the class of the object updated. For example, updating the age of some object other than an employee or subclass of employee, will not require the constraint `min_age` to be checked.

To update the state of any object, a set-method of that object must be called; this method is specific to the update performed, and the class of object on which it is performed. The events upon which integrity maintenance must be performed then map exactly onto the calls of these set-methods. The required catchment elements are therefore compiled into these methods. Relevant events are caught, whether they originate from an application program, a human interactively updating the database, or indeed from a trigger firing as the result of some other event elsewhere in the database.

It is important to note that the conceptual events which may activate a rule are more than simply the invocations of methods; they are declaratively specified changes in the state of the database; catchment elements are planted wherever necessary among the methods to detect these changes of state. The catchment elements are not the rules, which are separate objects, but objects which send a message to the appropriate rule. This approach, together with the automatic generation of the appropriate method code, circumvents problems associated with encoding a rule directly with a method, (listed in [DPG91]): since the rule is not present in the method, it is not necessary to touch these methods when, for example, examining the status

of a rule, or altering its definition[9]; since this code is automatically generated, the programmer does not need to know about how rules interact with methods.

### 4.2.1 Integrity Maintenance in General

A similar event-driven structure can be constructed for any of the constraints and triggers expressible in NOODL (with the exception noted above); a few details have been omitted for clarity. It may be necessary to include in a catchment, updates to instances of the transitive closure of subclasses of a class, rather than a single class. Further, the catchment may need to be enlarged to accommodate overriding in subclasses either of a rule itself, or of the properties it references. Further, since a rule may be defined differently over different classes, actual checking involves despatching on the class of the object which the event may affect under that rule.

On the other hand, since property obverses are automatically maintained by the system, half of the catchment elements can immediately be omitted without loss of security.

## 4.3 Automatic Generation of an Integrity Management System

In [BK92] a schema compiler is described, which reads a schema written in NOODL, and generates the data structures necessary to represent the application model which the schema represents. The user is given a set of procedures which query and manipulate the stored data, supporting the functionality of the model on which NOODL is based [Bar92]. The user has no need to know what exists behind this message interface, allowing physical data independence. Figure 9 shows the user's applications interacting with the persistent store through the automatically generated message interface.

This schema compiler is being modified to examine the terms appearing in rule specifications in a NOODL schema, and construct internal event-check tables analogous to that in table 1. It can then install the necessary

[9] More exactly, a change in definition which requires a new action, or requires a new predicate to be checked over the same configuration of objects, does not require the method to be touched. However, redefining the rule so that it applies to a different configuration of objects will require the catchment elements to be relocated. This extra work is the cost of allowing the greater generality of rules that may assert something about more than one object. In any case, the new method code can be generated automatically and recompiled independently of the rest of the system, so the overhead on rule evolution is not great.

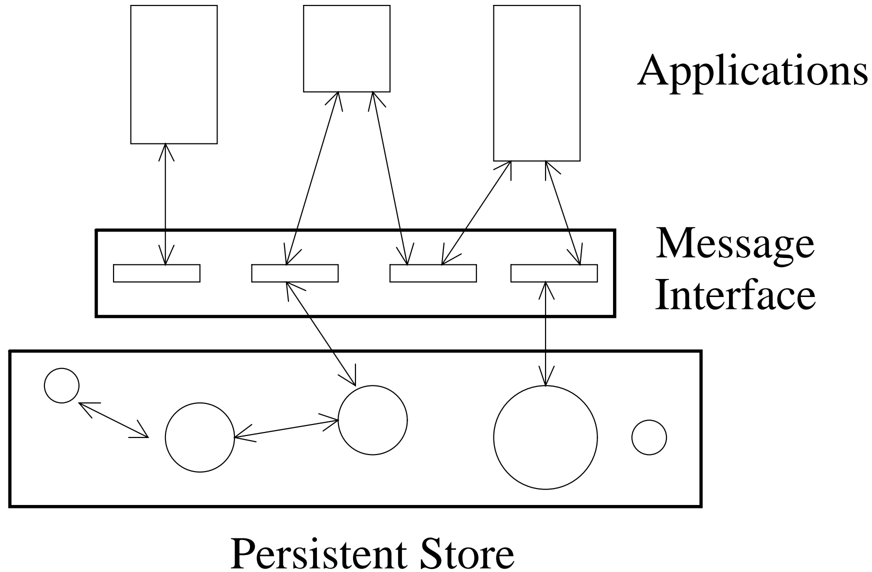


Figure 9: Semantically Secure Message Interface

catchments for the constraints and triggers defined in the schema. All the necessary information is available in the symbol tables of the schema compiler.

## 5 Some Related Work

Support for constraints in a persistent environment is examined by Cooper in [CQ91], where a general taxonomy of constraints is presented. This work explores what constraints can be expressed within the framework of various data models, but does not address active rules, nor strategies for management of integrity enforcement.

Owoso has described a variant of PS-algol [Mor88] where assertions may be made about language objects [Owo84]; a modified compiler attempts to check these assertions statically, and if this fails, compiles an appropriate check into the user's program. This provides a useful language extension, but the assertions provided are at a physical rather than conceptual level; again, active rules are outwith the scope of this work.

Hull describes an imperative language *Heraclitus* with a relational calculus sublanguage [HJ91], which includes *deltas* as first class values; a delta

is an object representing the change a given transaction would produce on the persistent store. Deltas can be combined, and their effects examined, without actually updating the store. This mechanism supports some 'active database' functionality in the language. However, only relations may persist, and deltas are explicitly manipulated in a procedural syntax by the user.

The integrity management constructs of section 3 are inspired by those of FCM [Sut90]. However, since concurrency is not an issue here, no constructs are required to manage it. Further, the idea of *default enforcement* has not been supported, but this causes no loss of expressivity since the enforcement status of any rule may persist within the IMS. We have extended Sutton's constructs from constraints also to Condition-Action rules, providing different coupling modes for rule execution, thereby integrating constraints and triggers within the same execution model. This has lead to a more orthogonal set of constructs than those of FCM, since the `clear` and `commit` operations may occur at any time; this permits, for example, trigger-firing to be suspended from one program execution to another.

## 6 Summary

Constructs for specifying semantic integrity in database schemata have been reviewed, and the realisation of these constructs in the modelling language NOODL described. We claim that although these constructs are intended as a tool for conceptual specification, knowledge (or assumption) of the execution model of their (possible) enforcement flavours their semantics; some guidelines for the use of these constructs are given. Further, another set of constructs have been presented, which allow detailed control over the management of integrity in a system of persistent applications. A system has been described which infers which events may compromise the integrity of which objects in the database serving these applications, and efficiently maintains integrity using a novel event-driven architecture. This system brings some 'active database' features to a persistent programming environment.

## References

[ACL91] Rakesh Agrawal, Roberta Cochrane, and Bruce Lindsay. On Maintaining Priorities in a Production Rule System. In Guy M

Lohman, Amilcar Sernadas, and Rafael Camps, editors, *proc VLDB 17*, pages 479 – 487, Barcelona, Sep 1991. Morgan Kaufmann.

[Bar92] Peter J Barclay. *Managing Complex Data*. PhD thesis, Napier Polytechnic, Edinburgh, 1992 (forthcoming).

[BCG$^+$87] J Banerjee, H-T Chou, JF Garza, W Kim, D Woelk, and N Ballou. Data Model Issues in Object Oriented Applications. *ACM Transactions on Office Information Systems*, 5(1):3 – 26, Jan 1987.

[BK91] Peter J Barclay and Jessie B Kennedy. Regaining the Conceptual Level in Object Oriented Data Modelling. In *proc BNCOD-9*, Wolverhampton, Jun 1991. Butterworths.

[BK92] Peter J Barclay and Jessie B Kennedy. Developing Persistent Application Systems. Technical report, Napier Polytechnic of Edinburgh, January 1992.

[Bor77] Alan Borning. ThingLab - An Object Oriented System for Building Simulations Using Constraints. *Proc 5th International Conference on AI*, 1977.

[CDKM89] QI Cutts, Alan Dearle, Graham NC Kirby, and CD Marlin. WIN: a Persistent Window Management System. Technical report, University of St Andrews, 1989.

[CK87] Quintin Cutts and Graham Kirby. An Event-Driven Software Architecture. Technical report, University of St Andrews, Oct 1987.

[Coc82] W Paul Cockshott. *Orthogonal Persistence*. PhD thesis, University of Edinburgh, 1982.

[Coo90] Richard Cooper. *On The Utilisation of Persistent Programming Environments*. PhD thesis, University of Glasgow, 1990.

[CQ91] Richard Cooper and Zhenzhou Qin. Constraint Management in a Configurable Data Modelling System. Technical report, University of Glasgow, Aug 1991.

[Dat87] CJ Date. *An Introduction to Database Systems.* Addison-Wesley, 1987.

[Day88] Umeshwar Dayal. Active Database Management Systems. In *proc 3rd International Conference on Data and Knowledge Bases*, pages 150 – 169, Jerusalem, Jun 1988.

[DCBM89] Alan Dearle, Richard Connor, Fred Brown, and Ron Morrison. Napier88 - A Database Programming Language? In *proc DBPL 2*, 1989.

[DPG91] Oscar Diaz, Norman Paton, and Peter Gray. Rule Management in Object Oriented Databases: a Uniform Approach. In Guy M Lohman, Amilcar Sernadas, and Rafael Camps, editors, *proc VLDB 17*, pages 317 – 326, Barcelona, Sep 1991. Morgan Kaufmann.

[EN89] Elmasri and Navathe. *Fundamentals of Database Systems.* Benjamin/Cummings, 1989.

[gen90] *Generis Users' Guide.* Deductive Systems Limited, 1990.

[HJ91] Richard Hull and Dean Jacobs. Language Constructs for Programming Active Databases. In Guy M Lohman, Amilcar Sernadas, and Rafael Camps, editors, *proc VLDB 17*, pages 455 – 467, Barcelona, Sep 1991. Morgan Kaufmann.

[Hug91] John G Hughes. *Object Oriented Databases.* Prentice Hall, 1991.

[Lel88] Wm Leler. *Constraint Programming Languages: Their Specification and Generation.* Addison Wesley, 1988.

[LW91] Peter Lyngbaek and Kevin Wilkinson. An Overview of the Iris Kernel Architecture. In Gordon Blair, John Gallagher, David Hutchison, and Doug Shepherd, editors, *Object Oriented Languages, Systems and Applications*, pages 328 – 347. Pitman, 1991.

[MBCD89] R Morrison, F Brown, R Connor, and A Dearle. The Napier88 Reference Manual. Technical report, Universities of Glasgow and St Andrews, Jul 1989.

[Mor88] Ron Morrison. PS-algol Reference Manual. Technical report, University of St Andrews, Feb 1988.

[Mos81] J Eliot B Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* PhD thesis, MIT, May 1981.

[Owo84] GO Owoso. *Data Description and Manipulation in Persistent Programming Languages.* PhD thesis, University of Edinburgh, 1984.

[pos90] *Postgres Reference Manual (version 2.0).* University of California, 1990.

[Pou88] Alexandra Poulovassilis. FDL: An Integration of the Functional Data Model and Functional Computational Model. In *proc BNCOD-6.* Cambridge University Press, 1988.

[SC91] ITA Spence and BN Carey. Customers do not want Frozen Specifications. *Software Engineering Journal*, pages 175 – 180, Jul 1991.

[SJGP87] Michael Stonebraker, Anant Jhingran, Jeffrey Goh, and Spyros Potamianos. *On Rules, Procedures, Caching and Views in Database Systems.* University of California, 1987.

[SRH] Michael Stonebraker, Lawrence A Rowe, and Michael Hirohama. *The Implementation of Postgres.* University of California.

[Sut90] Stanley M Sutton. FCM: A Flexible Consistency Model for Software Processes. Technical report, University of Colorado, Boulder, Mar 1990.

[TL82] Tsichritzis and FH Lochovosky. *Data Models.* Prentice Hall, 1982.

[WL89] Stephen P Weiser and Frederick H Lochovsky. OZ+: An Object Oriented Database System. In Won Kim and Frederick H Lochovsky, editors, *Object Oriented Concepts, Databases and Applications*, pages 309 – 337. Addison Wesley, 1989.