METASPEX

Metaspex Technical Insights

II - Referential Integrity

Introduction

Referential integrity is an old concept emanating from relational databases. Thanks to the addition of constraints in the database schema, its principle is that altering information in the database cascades to other pieces of information on disk, ensuring a simple form of information integrity. RDBMS referential integrity operates exclusively at run time and inside the database server. The code lying in the application interacting with the RDBMS, and the data it manipulates in memory, are left to the developer to take care of.

Metaspex, with the introduction of **own-a** and **link-to** families of relationships, substantially extends and refines the concept. Metaspex referential integrity applies equally to objects in memory and data lying in databases, even including across different database products. Metaspex promotes a model whereby the structure of information in memory and inside the database are both managed in a similar manner. Metaspex referential integrity actively maintains data integrity like RDBMS referential integrity, but it also performs checks.

Referential integrity checks have both static and dynamic aspects. Some of the properties of referential integrity are enforced at compile time, while some others are verified at run time. For instance, when an ontology declares an **own** relationship pointing at a **root**, the code will not compile.

The purpose of this document is to give developers clarity on Metaspex referential integrity and to offer Metaspex code samples highlighting it. Let's start with static referential integrity checks.

Static referential integrity checks

Consider the code below, declaring for now a simple ontology containing only one type, a booking:

```
#include "hx2a/root.hpp"

class booking: public root<>
{
   HX2A_ROOT(booking, "bkg", 1, root, ());
};
```

For now, we won't add any data in a booking, it's just a document containing only a **root** object. Compiling the ontology will succeed; Metaspex referential integrity won't object to the code above.

Let's now declare a user type (Metaspex comes with such a ready-made user type, you would not need to write this from scratch, so this is only for the sake of the example):

```
#include "hx2a/root.hpp"

class user: public root<>
{
   HX2A_ROOT(user, "us", 1, root, ());
};
```

Once again, this code will compile fine.

We assume that the booking type declaration is in the header file "booking.hpp". Let's now add an **own list** relationship from a user to their bookings:

```
#include "hx2a/root.hpp"
#include "hx2a/own_list.hpp"
#include "booking.hpp"

class user: public root<>
{
   HX2A_ROOT(user, "us", 1, root, (_bookings));

   own_list<booking, "bkgs"> _bookings;
};
```

An attempt to compile this code will fail. The reason is that referential integrity forbids a **root** type to be **owned**. It can only be the target of a **link** relationship. **Root**s appear at the top of a document only, and cannot appear somewhere inside. The ontology is incorrect.

Here are the compile-time checks performed by referential integrity:

- A **root**/**entity** type cannot be the target of an **own** relationship.
- An **element** cannot be the target of a **link** relationship.

Type safety, performed by the C++ compiler, could be listed among static referential integrity checks. They are naturally performed, given that Metaspex uses the strongly-typed C++ programming language as a host. Ontology types are C++ types.

Dynamic referential integrity checks

Static checks are essential and powerful, because they ensure that if your application is validated by the compiler, there are higher assurances that it will behave properly. However, not all checks can be made static. Some of them have to be performed dynamically.

Let's see some code:

```
#include "hx2a/root.hpp"
#include "hx2a/own.hpp"
#include "hx2a/components/address.hpp"

using namespace hx2a;

namespace my_ontology {
   struct user: public root<>
   {
      HX2A_ROOT(user, "us", 1, root, (_address1, _address2));

      own<address, "addr1"> _address1;
      own<address, "addr2"> _address2;
   };

   void f(const rfr<user>& u, const address_r& a){
      u->_address1 = a;
      u->_address2 = a; // Assignment of another own with the same address object.
   }
}
```

To make the code simpler, we are just reusing a type offered in the Metaspex Foundation Ontology, the address. It is an **element** which can be the target of an **own** relationship.

The **rfr** bit will be clarified in a subsequent document. Just assume for now that it's a smart pointer pointing at a user.

The code above will compile just fine. However, it won't run fine. Referential integrity will cause an exception to be thrown at the moment when the second assignment takes place. The reason for that is that a given object, here the address object, cannot be owned more than once.

To be clear, the reason for the second assignment to be rejected is unrelated to the fact that it's the same user object which is assigned the same address twice. The code below would fail too, even if the two users were different:

```
void f(const rfr<user>& u1, const rfr<user>& u2, const address_r& a){
   u1->_address1 = a;
   u2->_address2 = a;
}
```

It is the fact that the same address would become the target of two distinct **own** relationships which causes referential integrity to reject the assignment.

Note that this dynamic check applies only to **own** relationships. A given object which is an instance of a **root**, an **entity** or an **anchor** can be the target of multiple **link**s.

Dynamic referential integrity checks do not stop there. There are dynamic referential integrity checks performed at **link** establishment too. Consider the following code:

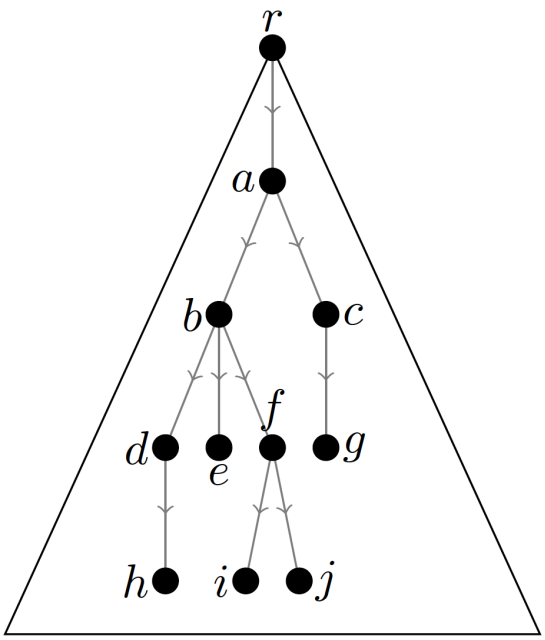
```
#include "hx2a/root.hpp"
#include "hx2a/anchor.hpp"
#include "hx2a/link.hpp"

using namespace hx2a;

namespace my_ontology {
   struct address: public anchor<>
   {
      HX2A_ANCHOR(address, "addr", anchor, ());
      address(){}
   };

   struct user: public root<>
   {
      HX2A_ROOT(user, "us", 1, root, (_address));

      link<address, "addr"> _address;
   };

   void f(const rfr<user>& u){
      u->_address = make<address>(); // Assignment of the link to a newly-made address.
   }
}
```

The **make** function will be described in a subsequent document. Assume for now that it builds a new address object, like C++ new would.

The code above compiles fine. However, the assignment in the function will fail at run time. The reason for that is that the address **anchor** object in the ontology is not transitively attached to a **root** object in a document.

A **link** can only be established towards an object participating in a whole document. If we can venture a philosophical explanation of that rule, it would be: “A **link** cannot be established towards information without full context.” Metaspex uses the word “context” to describe the origin object and the relationship towards a given object.

At this point in the document, instead of using code to illustrate principles, let’s use diagrams–it will be easier. The diagram below:



Graphical representation of a document

Represents a whole document as a triangle. At the top of the triangle, the black dot “r” is the **root** object of the document. It is an instance of a **root** or **entity** type in the ontology.

The arrows on the diagram represent exclusively **own** relationships. Every object that can be reached directly or indirectly from the **root** object is a participant in the document. As **own** relationships are exclusive, a given object, like “a,” “b,” or “c,” can only be in this document, and not in another one.

In the diagram above, the object “r” bears an **own** relationship towards object “a,” which, in turn, bears an **own** relationship towards objects “b” and “c,” etc.

The same diagram can be used whether the document is in memory, with its objects allocated on the heap, or somehow virtually inside the database. The diagram above will be used in the remainder of this document, applying equally whether the document is in memory or in the database.

A property of Metaspex is that at a given point in time, if one object in a document is in memory or in the database, all the other objects in the same document “live” also in the same place, respectively in memory or in the database. Documents are not “across” memory and database. They are acquired in a single I/O from the underlying database system to be brought to memory, and they are flushed entirely from memory to database when they are saved and evicted from the Metaspex cache.

Dynamic referential integrity automatic maintenance

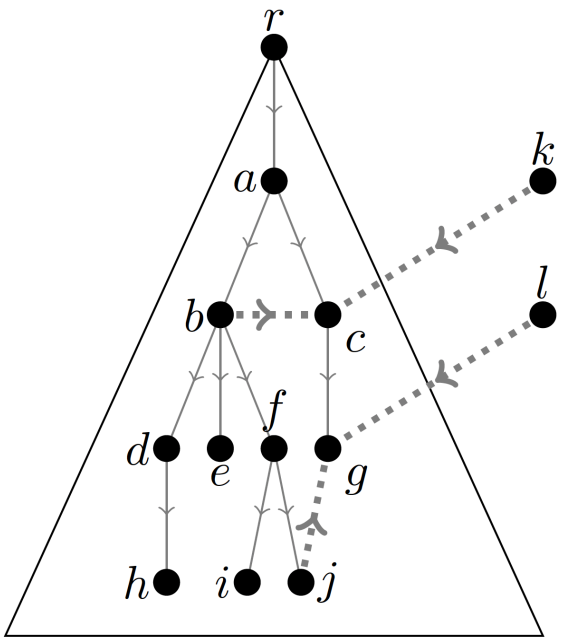
Metaspex documents come with an automatically generated API that supports such actions as unpublishing an entire document (removing it), cutting a sub-document from its whole document, etc. These APIs can introduce situations that would violate referential integrity if not “followed up.” In other words, if referential integrity were to check all the documents after modifications were made on them, there could be violations. To avoid this, Metaspex automatically maintains referential integrity by making additional modifications to documents, modifications that are implied and necessitated by the initial unpublish or cut action.

Here is an example: two documents are created, and a **link** is established from an object in the first towards a “deep” object in the second (“deep” meaning not the **root** object itself). Referential integrity dynamic checks allow that. Now the deep target object is cut (using the **cut** function generated automatically by Metaspex). Suddenly we have a **link** which exists from a document to an object which is not a participant in a whole document, as the target object has been cut from the **root** object it is transitively attached to with **own** relationships.

This is why dynamic referential integrity automatic maintenance exists. It ensures in general that the high-level API generated by Metaspex preserves referential integrity. In the specific example we discussed, it means that cutting a sub-document from its whole

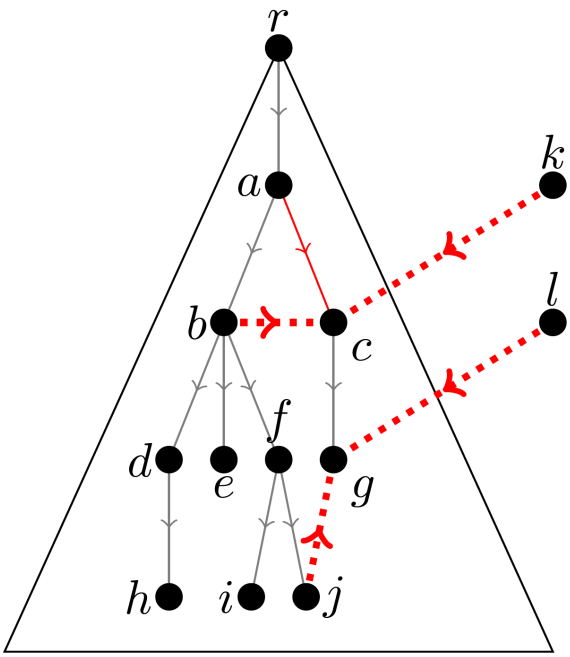
document should trigger an automatic break of the **link** relationships pointing at the object at the top of the sub-document. Not only that, it also cuts all the **link**s pointing at any object inside the sub-document.

This is illustrated by the following diagrams. Here is a document before an object is cut:



Here **link**s are represented by dotted lines. Objects “k” and “l” are in some other document.

Then the object “c” is cut using the Metaspex-generated function that every Metaspex ontology object instance possesses. The broken relationships are highlighted in red:



- The “a” to “c” **own** relationship is broken by the **cut**;
- The sub-document below “c” contains “c” and “g”;

- Every **link** external to the sub-document and pointing at “c” or “g” is also broken by automatic reference integrity maintenance.

If there were **link**s pointing at objects other than the sub-document under object “c,” they would be preserved by referential integrity. **Link**s within the sub-document and not pointing at “c” are preserved too.

In case a whole document has to be removed, **root**s offer an **unpublish** function which causes all the external **link**s pointing at objects inside the document to be cut by referential integrity. **Root**s can also be cut, with the same effect.

The various **own**- or **link**-based relationships offered by Metaspex allow control of the behavior of each relationship when referential integrity kicks in. Metaspex distinguishes “weak” and “strong” one-to-one relationships. When a weak **link** (**weak_link**) is broken by referential integrity maintenance, it is set to the null relationship. When a strong **link** (**strong_link** or simply **link**) is broken, the removal of the origin object of the **link** relationship is performed in cascade too.

Through this cascading maintenance process, **own** relationships are broken too. Weak **own** relationships (**weak_own**) are set to null, while strong ones (**strong_own** or simply **own**), in addition, propagate the process to the origin object of the **own** relationship, in cascade.

When **root** objects are reached, their corresponding document gets unpublished. At commit time, it will be removed from the database.

List relationships simply “shrink” automatically without any propagation to the origin of the relationships.

In comparison with RDBMS referential integrity, which fuses referential integrity constraints and the management of information lifespan, Metaspex referential integrity distinguishes two levels. The lifespan of objects in memory is completely independent from the structural operations performed automatically by referential integrity maintenance on documents. Objects are destroyed only when they are not used any longer, whether in a document or through a smart pointer.

Database considerations

We said earlier that Metaspex referential integrity operates both in-memory and in-database. To illustrate this, let’s take the example of an object being cut inside a document, and **link**s pointing at it. Let’s assume that there are documents inside one or

several databases bearing **link**s towards the cut object. These documents can be purely on disk, present in no application memory anywhere.

Metaspex implementation of referential integrity guarantees that referential integrity will be applied to these documents as well. It also guarantees that the operation will be performed without any secondary index being used.

Metaspex referential integrity will work whether the databases involved are the same as the one the cut object originally came from, different ones, different ones running in different database products (e.g., with **link**s across MongoDB and Couchbase), even different ones running across different datacenters if necessary.

A real-world example

We used a booking type example at the beginning of this document; now here is a real-world example of an application managing bookings. The code below uses both **link** and **weak_link**, each for a specific purpose controlling referential integrity.

```
class booking: public root<>
{
   HX2A_ROOT(booking, "bkg", 1, root, (_event, _host, _guest,
_messenger_participation, _note, _check_in_timestamp));
   // ...
private:
   // Strong link. The booking is removed if the event is removed.
   link<event, "e"> _event;
   // The booking is retained even if the host (the user who invited the guest)
is removed.
   weak_link<user, "h"> _host;
   // The booking is removed if the corresponding user is removed.
   link<user, "g"> _guest;
   weak_link<messenger::participation, "mp"> _messenger_participation;
   slot<string, "n"> _note;
   slot<time_t, "cit"> _check_in_timestamp;
};
```

The code above does not use “deep” objects. Each of the **link**s points at a **root**. Even in this simple case, referential integrity is of great value.

Conclusion

The coexistence of **own** and **link** families of relationships, offered for the first time by Metaspex, allows us to go far beyond the conventional definition of referential integrity, to give it a proper solid definition. The fact that RDBMS and graph databases have essentially only the lax **link** relationships and are devoid of **own** relationships prevents them from pushing the concept of referential integrity further. The same goes for document databases, which understand only the **own** relationships that stay within a document, but ignore the **link** relationships. Data models without **own**s have no spine, while data models without **link**s have no cohesion.

The introduction of the substantially enhanced definition of referential integrity in Metaspex offers a number of benefits:

- Making applications safer by ensuring that the enhanced definition of referential integrity is automatically enforced.
- Elevating abstraction to the level of “code as a specification,” removing a lot of tricky-to-write and tricky-to-debug code. This enhances both engineering productivity and applications’ quality.
- Opening up the opportunity to define enterprise components that come with their own behavior, making them reusable across applications and their ontologies.
- Having all the above independent from the implementations of underlying databases.
- The implementation of referential integrity is achieved without any use of indexes, turning referential integrity into a fast and efficient process.
- A surprising byproduct of referential integrity is the automatic destruction of ontology objects. The algorithm is original to Metaspex, and represents an alternative to both reference counting, which imposes limitations, and garbage collection, which imposes high costs. Like garbage collection, it allows cyclical references, but like reference counting, it is synchronous. It is a “best of both worlds” solution.