METASPEX

Metaspex Technical Insights

I - Ontology

Introduction

Metaspex promotes a data model centered way of developing real-time business applications. In contrast, the traditional approach relies on relational database schemas with cumbersome mappings between relational tables and application data structures, mappings which necessitate a great deal of translation code.

To foster progress–substantially better productivity when engineering business applications, safer and more sophisticated solutions, and far better performance–Metaspex elevates abstraction. The first area where Metaspex does so is at the heart of business applications: data modeling. Metaspex moves beyond database schemas to provide the power of an enhanced, original data modeling technique: Metaspex ontologies.

Metaspex ontologies are already powering back-end development with radically new efficiency. Let's look at how this works.

From Wikipedia (*Ontology (information science*)): "An ontology encompasses a representation, formal naming, and definitions of the categories, properties, and relations between the concepts, data, or entities that pertain to one, many, or all domains of discourse. ***More simply, an ontology is a way of showing the properties of a subject area and how they are related, by defining a set of terms and relational expressions that represent the entities in that subject area*** **[emphasis added].**"

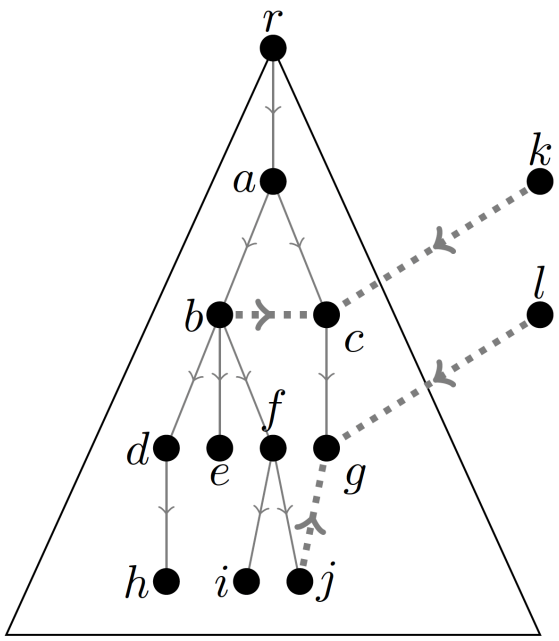
Unlike the Semantic Web interpretation of ontologies, which focuses on the first part of Wikipedia's definition, and revolves around high-level standards for information exchange, Metaspex uses ontologies in the more philosophically accurate sense conveyed by the second part of Wikipedia's definition. Metaspex uses ontologies to describe reality, the "domain in itself," as a new and improved way to do data modeling, independently from information exchange, as a replacement for core operational database schemas. It offers its own precise definition and implementation of ontologies with additional definitions unifying concepts drawn from relational, graph and document data modeling. We come from implementation, not standardization, so Metaspex concerns focus specifically on bottom-up abstraction elevation rather than top-down normalization.

The purpose of this document is to show how ontologies are concretely defined with Metaspex code, within the Metaspex host programming language: C++.

Trees and links

Metaspex considers that the data of a domain being modeled consists of trees of objects, that we call “documents”. In addition to these trees of objects, Metaspex enables **link**s within documents, from object to object, and also between objects in different trees. **Link**s can point at the **root** of a document, but also at objects deep inside documents.

This is very similar to the Web, where HTML pages are tree structures, elements containing other elements, and URLs establish links between elements, within the same page or between pages.



The diagram above shows a tree of objects (the black circles). This tree is called a “document.” It is represented by a triangle. “r” is the **root** object of the document, “a” to “j” are objects inside the document, while “k” and “l” are objects in one or two distinct documents. When these objects are in memory, they are independently allocated on the heap.

The hierarchical relationships between objects of the same documents are called **own** relationships. They are shown as solid arrows pointing down. **Link**s are represented by dotted lines. **Own**s and **link**s are both oriented.

From an ontological standpoint, the location of documents in databases is irrelevant. It is an implementation choice relegated to configuration.

Slots

Objects in a Metaspex ontology can also bear atomic attributes, integers, strings, booleans, enumerated values, timestamps (time_t), etc. Metaspex offers an abstraction to describe them (we'll see later why). These attributes are called **slot**s.

Roots, elements and anchors

Every type in a Metaspex ontology belongs to one and only one category: **root**, **element** or **anchor**. The reason for that is that Metaspex offers abstraction to manage them.

- **root**s are types which appear only at the **root** of a document;
- **element**s are types which can only be the target of **own** relationships;
- **anchor**s are types which can be targets of both **own** and **link** relationships.

Referential integrity rules apply to these types, but that is the topic of another paper in this series. The only constraint worth mentioning for now is that **own** relationships between objects are exclusive. A given object can be the target of at most one **own** relationship at any given point in time.

Here is the definition of a Metaspex ontology containing only one **element** type, an address:

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

using namespace hx2a;

class address: public element<>
{
   HX2A_ELEMENT(address, "addr", element, ());
};
```

The address type is empty for now. The code above compiles as is with an off-the-shelf C++ compiler such as g++ or clang without any preprocessing other than the C++ preprocessor.

The header file "hx2a/element.hpp" comes from Metaspex. The address type inherits from Metaspex's **element** type located in that header file. This is enough to specify that an address is an **element** in the sense of Metaspex ontologies. As such, an address can be the target of **own** relationships.

We won't elaborate for now on the HX2A_ELEMENT macro used. The macro will become clearer when we talk of database persistence later. Suffice to say that the first argument of the macro is the type being defined and the third is the base type (here, **element**).

We won't be able to create address documents, because an **element** cannot appear at the **root** of a document. Let's add another type, a user:

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

using namespace hx2a;

namespace my_ontology {
   class address: public element<>
   {
      HX2A_ELEMENT(address, "addr", element, ());
   };

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

We have added:

- Metaspex's header "root.hpp";
- A namespace to tie together our ontology;
- The user type, inheriting from **root**;
- A call to the HX2A_ROOT macro, which takes an additional argument (1) we're not going to talk about yet.

This code compiles fine as is.

For now, no relationship has been declared between the user and the address types. Let's add it. As address is an **element**, it can be the target of **own** relationships. We're going to describe that a user document contains the user's address. We do it this way:

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

using namespace hx2a;

namespace my_ontology {
   class address: public element<>
   {
      HX2A_ELEMENT(address, "addr", element, ());
   };

   class user: public root<>
   {
      HX2A_ROOT(user, "usr", 1, root, (_address));
      own<address, "a"> _address;
   };
}
```

We have added the "own.hpp" header file from Metaspex. We have used the **own** abstraction inside the user class to indicate that a user owns an address. The additional _address member is listed between the parentheses coming at the end of the HX2A_ROOT macro. For now, let's put aside the second argument of the **own** template. Its purpose will become clearer when we talk of persistence.

If we had created a **link** between a user and an address, the code would not have compiled. An address is not an **anchor**, it is just an **element**. This check is part of static referential integrity, which is a topic described in another document.

Also, we haven't added any constructors for the address and user types. We'll see that in another document.

We can add **slot**s in a very similar way:

```
#include <string>

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

```
#include "hx2a/own.hpp"
#include "hx2a/root.hpp"
#include "hx2a/slot.hpp"
using namespace std;
using namespace hx2a;

namespace my_ontology {
   class address: public element<>
   {
      HX2A_ELEMENT(address, "addr", element, (_number, _street, _city, _state,
_zipcode));
      slot<unsigned int, "num"> _number;
      slot<string, "strt"> _street;
      slot<string, "cty"> _city;
      slot<string, "st"> _state;
      slot<string, "zip"> _zipcode;
   };

   class user: public root<>
   {
      HX2A_ROOT(user, "usr", 1, root, (_name, _address));
      slot<string, "nm"> _name;
      own<address, "a"> _address;
   };
}
```

We included the "slot.hpp" header file, we added a few **slot**s to the address type, listing them between the parentheses in the HX2A_ELEMENT macro, and we added a **slot** in the user type.

Sometimes, types cannot be listed in a specific order where they are declared in full first and used later on. The code above can be adjusted to cater to that:

```
#include <string>

#include "hx2a/element.hpp"
#include "hx2a/own.hpp"
#include "hx2a/root.hpp"
#include "hx2a/slot.hpp"

using namespace std;
using namespace hx2a;

namespace my_ontology {
   class address; // Forward declaration of address.

   class user: public root<>
   {
      HX2A_ROOT(user, "usr", 1, root, (_name, _address));
      slot<string, "nm"> _name;
      own<address, "a"> _address;
   };

   class address: public element<>
   {
      HX2A_ELEMENT(address, "addr", element, (_number, _street, _city, _state,
_zipcode));
      slot<unsigned int, "num"> _number;
      slot<string, "strt"> _street;
      slot<string, "cty"> _city;
      slot<string, "st"> _state;
      slot<string, "zip"> _zipcode;
   };
}
```

Here we have made a forward declaration of the ontology types, and used them when the full types are defined. The code compiles fine this way.

Also, when an ontology becomes very large, it is possible to break down the ontology file into separate header files, for instance one header file per ontology type. That way, we could have one header file for “user” and another one for “address.”

Links

We have used only an **own** relationship. We haven't defined a **link** yet. Let's do it. To keep code short, we assume that the type organization has been defined in a separate header file, and it is a **root**. We can express that a user belongs to an organization the following way (we here omit the **slot**s for the sake of conciseness):

```
#include "hx2a/element.hpp"
#include "hx2a/link.hpp"
#include "hx2a/own.hpp"
#include "hx2a/root.hpp"

#include "my_ontology/organization.hpp"

using namespace hx2a;

namespace my_ontology {
   class address: public element<>
   {
      HX2A_ELEMENT(address, "addr", element, ());
   };

   class user: public root<>
   {
      HX2A_ROOT(user, "usr", 1, root, (_organization, _address));
      link<organization, "o"> _organization;
      own<address, "a"> _address;
   };
}
```

If the type "organization" were an **anchor**, the code above would still be correct and would compile fine. An **anchor** is defined exactly like an **element**, except that it uses the "anchor.hpp" header file, the **anchor** base type and the HX2A_ANCHOR macro.

List relationships

The **own** and **link** relationships we've seen so far are so-called one-to-one relationships. These are only a subset of the relationships necessary to define a real-life ontology. Specific domains are full of one-to-many relationships. Take for instance a survey, made of a list of questions. If you want to specify a survey ontology, you need to express that the survey type, a **root**, will bear a one-to-many relationship to the question type.

This is why Metaspex offers both **own** and **link** list relationships. See below an example of a survey ontology:

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

using namespace hx2a;

namespace survey_ontology {
   class question: public element<>
   {
      HX2A_ELEMENT(question, "quest", element, ());
   };

   class survey: public root<>
   {
      HX2A_ROOT(survey, "srv", 1, root, (_questions));
      own_list<question, "qs"> _questions;
   };
}
```

We have included the "own_list.hpp" Metaspex header and used it inside the survey type.

Note that list relationships have a specific behavior when dealing with referential integrity. Consult the document dedicated to referential integrity for details about it.

Both **own_list** and **link_list** behave similarly to C++ std::list. They support iterators and algorithms.

Vectors

Similarly, Metaspex offers one-to-many vector relationships as **own_vector** and **link_vector**. They behave similarly to C++ std::vector.

Similarly for **slot**s: **slot_vector** is provided too.

Weak relationships

Metaspex offers variants of **own** and **link** relationships, which are “strong relationships” and aliases of **strong_own** and **strong_link**. The “weak” variants are **weak_own** and **weak_link**.

Their behavior differs substantially when automatic referential integrity maintenance kicks in. It is not the object of this document to detail that; please refer to the specific document on referential integrity for explanations of weak relationships.

Inheritance and polymorphism

Once you have Metaspex ontologies, which combine relational, graph and document modeling, you have access to powerful tools which provide opportunities to express sophisticated data modeling. As Metaspex data modeling extends to more complex cases, you soon find that you needed inheritance and polymorphism all along.

The survey example we just looked at is a good example. A survey is composed of a list of questions. However, question types vary a great deal and share very little from one to the other. The reality of surveys is that question types are very diverse. The survey industry is known to struggle with the addition of new question types. This is because the mapping to relational databases is very tedious.

We can illustrate how to resolve this problem very easily in the survey example by adding inheritance to the former specification:

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

using namespace hx2a;

namespace survey_ontology {
   class question: public element<>
   {
      HX2A_ELEMENT(question, "quest", element, ());
   };

   class single_choice_question: public question
   {
      HX2A_ELEMENT(single_choice_question, "scquest", question, ());
   };

   class multiple_choice_question: public question
   {
      HX2A_ELEMENT(multiple_choice_question, "mcquest", question, ());
   };

   // ...

   class survey: public root<>
   {
      HX2A_ROOT(survey, "srv", 1, root, (_questions));
      own_list<question, "qs"> _questions;
   };
}
```

We now have a base type, question and multiple derived types, each inheriting from question. Derived types use the HX2A_ELEMENT macro supplying the base type, question, as the third argument.

The **own_list** relationship on the survey type is now polymorphic. It points at the base type. On a survey document instance, the target question objects are actually instances of any of the derived question types.

Metaspex will handle this perfectly, including when handling persistence.

Note that Metaspex supports only single inheritance.

Inheritance unleashes capabilities which can be applied to almost any domain. Think of documents containing multiple messages, Internet of Things rules with multiple actions (sending an SMS, an email, calling a Web service, etc.). Inheritance is everywhere.

Reuse

A valuable property of Metaspex ontologies is the fact that once a type or even an entire ontology has been defined, it can be reused in other ontologies, by just grabbing types to compose new ontologies. Reuse can be done through:

- Pointing through **own** or **link** to an already-defined type;
- Inheriting from that type.

This compounds the productivity granted by the high level of abstraction offered by Metaspex constructs to define new ontologies. You can take a little bit of extra time to define types more carefully, and they become available as enterprise components to define new ontologies even faster.

Take the address we defined above:

```
#include <string>

#include "hx2a/element.hpp"
#include "hx2a/slot.hpp"

using namespace std;
using namespace hx2a;

namespace my_ontology {
   class address: public element<>
   {
      HX2A_ELEMENT(address, "addr", element, (_number, _street, _city, _state,
_zipcode));
      slot<unsigned int, "num"> _number;
      slot<string, "strt"> _street;
      slot<string, "cty"> _city;
      slot<string, "st"> _state;
      slot<string, "zip"> _zipcode;
   };
}
```

We could have defined it more carefully, we could have added a country code, we could have made this country code ISO 3166-compliant.

That is why Metaspex comes with a Foundation Ontology. It contains reusable enterprise components which allow defining new ontologies by simply grabbing predefined types and using them in your ontology. You can do that with your own types, and keep a growing library of enterprise components for your domain.

The address above is precisely one of the reusable components offered by the Metaspex Foundation Ontology. And it incorporates the ISO 3166-compliant country code we just talked about. No need to define it, just grab it. The same goes for the user or the organization types.

Metaspex comes with a Foundation Ontology which includes everything around security (including not just users and organizations, but also affiliations, roles, etc.). It also includes predefined HTML-5 compliant geolocations, matrices, time periods, time series for finance, survey components, messaging (as in Discord or Slack), IoT components such as device rules, all country codes, all languages, all currencies, etc. It keeps growing.

Ontology types are standard C++ types

Because Metaspex uses C++ as the host language, it benefits from the powerful C++ zero-overhead abstraction capabilities. Most of Metaspex abstractions are C++ templates. As a matter of fact, Metaspex is mostly 300,000 lines worth of C++ templates.

Metaspex's choice of C++ as the host language offers you innumerable benefits. You can use regular C++ code in ontology types; for instance, you can add member functions. Also, Metaspex ontologies are compiled using a powerful off-the-shelf C++ compiler, such as g++ or clang.

The code generated benefits from the vast amount of optimizations these compilers are capable of, in particular their ability to "collapse" multiple layers of abstraction and produce dense binaries which are very optimized. Metaspex applications, as a result, are usually made of a few tens of megabytes which can run from some of the smallest machines (e.g., a $15 Raspberry Pi Zero 2) to the largest ones (128-core public Cloud machines, or even bigger private machines).

Another benefit is that you can use basically any C++ construct around ontology types. You can add member functions. You can use templates. Ontology types can be generic, and the ontologies described can be generic ontologies. This is especially interesting when defining reusable enterprise components to make them even more powerful. As an example, the targets of relationships can be template parameters. The Metaspex Foundation Ontology makes use of these features.

A full knowledge of the complex C++ language is by no means necessary to use Metaspex to produce sophisticated business applications. Using Metaspex requires only a small subset of the C++ programming language.

Leveraging the full power of Metaspex ontologies

By experience it takes time to realize the full potential of Metaspex ontologies. If you have been used to the relational model, you might be tempted to replicate it within Metaspex ontologies. If you do so, you'll not benefit from what Metaspex can bring. The adjustment takes time.

For instance, if you are used to the fact that every structured piece of information is a table, you might be tempted to define every Metaspex type as a **root**. If you do so, you won't benefit from performance gains, and even more importantly, you won't get referential integrity. The latter is described in a dedicated document.

To a lesser extent, the temptation could be to define every non-**root** type as an **anchor**. After all, an **anchor** is more powerful than an **element**. It can be the target of **link**s, while an **element** cannot. This is not a good idea because there are benefits in compile-time referential integrity checks. Some types should not be the target of **link**s, and the compiler should be allowed to flag that misuse. But also, **anchor**s are slightly more expensive for Metaspex, so **element**s are a more economical alternative.

Persistence

It is now time to shed some light on some mysterious bits of the ontology code samples we used. Take for instance:

```
HX2A_ELEMENT(address, "addr", element, ());
```

We did not talk of the string coming as the second argument of the macro. Similarly, here, the second argument of **own**:

```
own<address, "a"> _address;
```

Or here:

```
slot<unsigned int, "num"> _number;
```

Metaspex calls these strings **tag**s. They are used for database persistence purposes. On a type, for instance in the HX2A_ELEMENT macro above, the string "addr" must identify uniquely the type address in the entire application. When you choose it, you must make sure of that. Failure to ensure that uniqueness will cause the application to fail to start. Using prefixes in these strings is a good idea to ensure that no collisions happen. If you define a survey ontology, it is a good idea to prefix these type **tag**s with "survey_", for instance.

A similar uniqueness is enforced for members, relationships and **slot**s. The second argument of the **own** and **slot** templates above must identify uniquely the corresponding type member across its entire inheritance line. Failure to ensure uniqueness will prevent the application from starting. Metaspex will check that.

To save documents in a database, ontologies do not need anything additional to what we have seen so far. Only the **tag**s will be used. They will be used in Couchbase to save JSON documents. The member **tag**s will be used as JSON keys. Similarly, the **tag**s are used when MongoDB is used to save documents as BSON keys.

When Metaspex documents are saved, they are saved with a one to one association with documents in document databases. They become an entire JSON document in Couchbase or an entire BSON document in MongoDB. Metaspex documents are never “half way” across database documents. No costly and corner-cutting object-relational mapping (“ORM”) is used. A document is acquired in one I/O and saved in one I/O.

Persistence is entirely automatic, no serialization and deserialization functions need to be written, there is no explicit save or acquisition of documents, and no database API is used ever, not even some query language (e.g., SQL++ with Couchbase or MongoDB query language). NoSQL means No SQL whatsoever. How this is achieved is the subject of subsequent technical insights.

Links are serialized and deserialized automatically. They can be within the same database instance (bucket), across buckets, or across databases, database products or even data centers.

The choice of databases is made purely in a configuration file read at application start by Metaspex. The binary of the application can be deployed on various databases, it just requires a restart to operate on another database or database product. Ontologies are completely identical across databases, and they are devoid of any trace of persistence.

Every **root** object comes automatically with metadata, and is assigned a unique identifier (a standard random 128-bit UUID).

Every member abstraction, relationship or **slot** comes with an automatic sensitivity to changes. It means that to describe a process updating an address for instance, the process will just have to assign a value to the corresponding **slot**. Metaspex will detect the change, and will save the modified overall document containing the address at commit time. It won't try to save documents which were not changed.

Relationships can be crossed simply using the "->" C++ notation, Metaspex will take care to ensure that the target object is retrieved properly, including when the relationship is a **link**. In case of a **link**, Metaspex will keep the target document in the application cache, and that cache will be flushed at the end of the transaction.

Please consult the technical insight on making objects and pointing at them for further details.

Ontology evolutions

There is another macro parameter we did not talk about. It's the third parameter of the HX2A_ROOT macro:

```
HX2A_ROOT(survey, "srv", 1, root, (_questions));
```

We can read "1" here. It is the version number of the document. Not just of the survey type, but of the overall document, including all the objects which are instances of types that can be reached transitively by **own** relationships from the survey type.

Every time one of the types which can participate in the corresponding document changes (typically adding members or removing them), the version number must be increased. The precise rules about version changes are described in the Metaspex reference guide. Some changes tolerate keeping the same version number.

Metaspex will automatically upgrade older version documents, following the same rules as the ones expected from JSON documents (even if JSON is not used, for instance with MongoDB and its BSON format). The upgrades can happen on the fly (e.g., when retrieving a document), or through a batch program that can be automated with Metaspex tools.

One to few and one to many

In the vast majority of cases, the relational model considers that relationships are potentially one to many, and when a piece of information “contains” other pieces of information, a relationship is created between the second, pointing at the first. This inversion, which is not so natural, allows “scale.” You can have millions of instances of the latter pointing at a single one of the first.

Metaspex ontologies offer an additional choice. You can distinguish between relationships which are “one to few” and the ones which are “one to many.”

While “one to many” can cater to the needs of scale, “many” meaning potentially millions, “one to few” caters to performance and database I/O optimizations. A template such as **own_list** can be chosen in case the origin of the relationship and its target should be in the same document, and should be saved and acquired in a single efficient I/O. If you take the example of a survey containing questions, we’re clearly in the case of a “one to few” **own** relationship. This is why we chose the **own_list** construct offered by Metaspex.

All this means that with Metaspex you finely control the granularity of documents and the corresponding I/O performance. In that sense, the abstractions offered by Metaspex are “good abstractions” as they do not mask subtleties, but offer both conciseness and productivity, without hampering performance. On the contrary, performance overall is much better than the performance offered by RDBMS, making document databases shine.

The cost of distinguishing between one to few and one to many is the intellectual challenge of defining things properly. Metaspex eliminates repetitive or boilerplate code. It removes toil, it does not eliminate thinking.

Multitenancy

An interesting property of some applications is multitenancy. Multitenancy allows economies of scale, because independent groups of users are insulated from one another, while using the same infrastructure efficiently. For software vendors, this means improved margins. For customers, it means more economical solutions.

Metaspex supports multitenancy out of the box. The only change required is to replace the **root** abstraction with the **entity** one, defined in the "hx2a/components/entity.hpp" header file. Metaspex will ensure that data is completely insulated, while being hosted in the

same databases. As a developer you can still describe use cases where you have processes across different tenancies, of course.

A subsequent document will provide more details on multitenancy.

Conclusion

Metaspex promotes a model whereby enhanced data modeling ensures developer productivity and safer applications. It greatly increases expressive power, including support for inheritance and polymorphism, while making no performance sacrifice.

As a matter of fact, Metaspex ontologies, because they are implemented with a one to one correspondence with document databases and because they are compiled with an efficient C++ compiler, offer unparalleled performance and operation cost.

By elevating abstraction through layered C++ templates, Metaspex eliminates almost all repetitive code. Elevating abstraction is a bounded process, as, at some point, the code is akin to formulating requirements. As these requirements are automatically and efficiently implemented using a regular C++ compiler, requirements also constitute technical documentation.

As a result, the Metaspex approach can be summarized as:

Requirement is code is technical documentation.