EMF Temporality

Jean-Claude Coté

Éric Ladouceur

1 Introduction

Things change. If we store information about the world this may not be a problem. After all when something changes one of the great values of a computerized record system is that it allows us to easily update a record without resorting to liquid paper or retyping pages of information.

Things get interesting, however, when we need to record the history of the changes. Not just do we want to know the state of the world, we want to know the state of the world six months ago. Even worse we may want to know what two months ago we thought the state of the world six months ago was. These questions lead us into a fascinating ground of temporal patterns, which are all to do with organizing objects that allow us to find answers to these questions easily, without completely tangling up our domain model. Of all the challenges of object modeling, this is both one of the most common and most complicated. (Taken from Martin Fowler, Patterns for things that change with time, http://www.martinfowler.com/ap2/timeNarrative.html)

1.1 Dimensions of Time

Just as I've described above, time is a pretty challenging notion in modeling. However I've skipped over the most awkward aspect of temporal models. We've all learned, if only from bad Science Fiction books, that time is the fourth dimension. The trouble is that this is wrong.

I find the best way to describe this problem is with an example. Imagine we have a payroll system that knows that en employee has a rate of $100/day starting on January 1. On February 25 we run the payroll with this rate. On March 15 we learn that, effective on February 15, the employee's rate changed to $211/day. What should we answer when we are asked what the rate was for February 25?

In one sense we should answer $211, since now we know that that was the rate. But often we cannot ignore that on Feb 25 we thought the rate was $100, after all that's when we ran payroll. We printed a check, sent to him, and he cashed it. These all occurred based on the amount that his rate was. If the tax authorities asked us for his rate on Feb 25, this becomes important.

In fact we can think that there are indeed two histories of Dinsdale's pay rate that are important to us. The history that we know now, and the history we knew on February 25. Indeed in general we can say that not just is there a history of dinsdale's pay rate for every day in the past, but there is a history of histories of Dinsdale's pay rates. Time isn't the fourth dimension; it's the fourth and fifth dimensions!

I think of the first dimension as **actual time**: the time something happened. The second dimension is **record time**, the time we knew about it. Whenever something happens, there are always these two times that come with it. Dinsdale's pay raise had a actual date of Feb 15 and a record date of March 15. Similarly when we ask what Dinsdale's pay rate was, we really need to provide two dates: a record date and an actual date. (Taken from Martin Fowler, Patterns for things that change with time, http://www.martinfowler.com/ap2/timeNarrative.html)

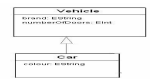
These quotes from Martin Fowler's Temporal Patterns illustrate the two flavours of temporality. The first pattern handles only actual time and assumes that record time is always the current time. The second pattern

enables you to specify a record time. These patterns are known as Temporality and Bi-Temporality. In our design we will use simple Temporality. However we will indicate how we could extend this design to support Bi-Temporality.

This document is divided into two sections. In the first section, we will review the current Andromeda implementation of Temporality. We will then present the proposed EMF implementation.

3 Proposed EMF implementation

To illustrate our proposed solution for EMF, we will first take a non-temporal model and add temporality to it. Here's an example of a non-temporal model.



3.1 Modeled Persistence

In this section, we will show how to add temporality to an object. We will do this in the modeling domain of EMF. By doing it this way, we leverage the persistence layer of EMF. We don't have to worry about it ourselves.

Suppose we want to add temporality to the car defined above. To do this, we add a temporal base class to the Car. Unlike Java, EMF supports double inheritance. This adds two new attributes to the Car: a date and a versionHolder. It also adds tree operations, which will be described later.

We can then specify which attribute of the Car we want to keep a history of by adding an annotation to the attribute. Specifying that an attribute is temporal will keep track of the different values this attribute took over time. In the case of a reference, it will keep track of which EObject the temporal reference is pointing to. But it does not say anything about the temporality of the referenced object. It may or may not be temporal.

This means that, if you have a Car with a reference to an Owner, the Car will keep track of which Owner object the Car references. However, it does not say anything about the state of the Owner. The Owner may be temporal or not.

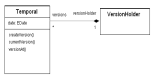
If you want the object you are referencing to be temporal, you need to use the same process. So, if the Car has a reference to an Owner and you want the Owner to be temporal as well, you'll have to model the Owner as inheriting from Temporal and specify which attributes of the Owner you would like to be temporal.

3.2 Modeled Temporal API

In this section, we will show what the tree modeled operations of the Temporal base class do and how to use them.

3.2.1 Temporal EObject

The Temporal EObject has a versionHolder attribute. The VersionHolder has a list of Temporal EObjects, which it uses to keep track of all the versions. Every version of a given Temporal object shares this VersionHolder.



Here's the object model. It consists of the group of versioned objects (Temporal objects with a date attribute) all pointing to the same VersionHolder. The VersionHolder has an ordered list of versioned car objects.

The Temporal EClass declares tree operations: createVersion, versionAt and currentVersion.

The createVersion(date) operation creates a new versioned car at the given date. The versioned car is assigned the given date and is added to the versions attribute of the VersionHolder.

The versionAt(date) operation returns the versioned car at the given date. It looks it up in the VersionHolder's versions attribute.

The currentVersion() operation returns the current car. It is not really a versioned object. It represents the Temporal Wrapper from the original Andromeda implementation. The current and versioned objects behave exactly the same. The only difference between them is that the current object simply has its date attribute set to the current date.

All the getters and setters performed on these versioned cars are intercepted and may be delegated to the VersionsHolder. Please refer to the pseudo code below for more details. Note all the versioned cars refer to the same VersionHolder. One Temporal has many versions all versions refer to a VersionHolder object, which in turn references all the versions. It is a many to 1 relation.

Notice that, when you version a car, it does not version the wheels collection it has or any of the wheels in the collection. The user has to explicitly do so. We will probably provide a utility to version a graph of objects. But the idea remains that each Temporal object is versioned independently, giving the user the maximum control over what gets versioned.

3.2.2 Future Versions

We have seen how you can make changes in the past of a Temporal object, but what about making changes in the future? This may sound complicated, but is in fact already handled in the proposed solution. Using the createVersion(date) operation, it is possible to create a version at any point in time, past or future. So, say you are told today that twenty years from now the car will run only on solar power so you have to set the consumption to 0 twenty years from now (2007). All you really have to do is create a version with a date of 2027 and set the consumption to 0. When the current date is 2027, the specified change will become visible to the current version object.

3.2.3 Time Granularity

In the current Andromeda implementation, the granularity of the date/time is to the millisecond. In the EMF implementation, we should consider offering the possibility of setting a coarser granularity. For example, by

the hour. The versions would then work by the hour. The creation of a version would return an existing version if one exists within an hour of the specified time. This could be easily achieved by subclassing the java.util.Date and making use of this DateTime subclass in our implementation. There's an example of this pattern on Martin Fowlers website.

3.2.4 Automatic Versioning

In this section we will show how using the basic building blocks above we will implement an automatic versioning feature. This feature removes the burden of versioning objects manually by calling createVersion yourself. It requires the concept of a central current date or working date. Once this central date is specified, the system reacts by giving the user a view into the system at the specified date.

As we have seen, the versioned objects hold on to a date. This date represents their view into the temporal data and is unchangeable. When a getter is called, the value returned is the value corresponding to the date of the versioned object. When a setter is called, it changes the value of the given attribute at the date of the versioned object. There is no concept of central date when working with the versioned objects.

Things are handled differently in the current version object (or moving version object). This object is aware of the current date. It has no date per say. It uses the date stored in the central location. So, by setting the central date, you can affect what a moving version object sees.

The current version object does not store any data either. It delegates this job to versioned objects. The current version objects use the central date to lookup the corresponding versioned object. When a getter is called on a moving version object, it calls the VersionHolder.versionAt(centralDate) method to retrieve the corresponding version. The returned version object is the version closest to the date specified. The moving version object then delegates the getter to the returned version object.

In the case of a setter, things are a little more involved since we have to consider the fact that we may not have a version object corresponding to the central date. If we do not have one, we must create one.

So, if the versioned object returned by the VersionHolder.versionAt(centralDate) method does not fall within the range of the central date, then we create a new versioned object (VersionHolder.createVersion(centralDate)) and then perform the setter with this newly created versioned object instead. If we do find a corresponding version. we simply use it.

3.3 Persistence Details

So far, we have not talked about how the actual data of the temporal versions is represented. In this section, we will show how the data is represented for the Temporal objects. We will use EMF-modeled classes to represent the data so that the implementation of temporality works across any kind of EMF resource. We will use the Temporal Object pattern to keep track of the versions.

Note: We have explored the possibility to use the "unsettable" property that EMF Features have. However, the "unsettable" property did not suit our needs since we need to keep the value and remember if it is set/unset.

Our strategy for representing temporal attribute data is to keep the information right in the Temporal objects. However, to eliminate the propagation issue, we will conceptually add a flag to every attribute to know whether a given attribute was set or not in the versioned object. We will see later how this flag is actually represented. Here's an example. Say we create the following versions:

Notice how with the 1950 car we did not change the consumption. The consumption attribute is marked (untouched). This enables us to propagate changes made to a Temporal object up to the time when it is changed explicitly.

The propagation happens when a value is set. Suppose we go back and change the 1930 car consumption to 9 km/l. To propagate the change to the 1950 car, all we have to do is set the consumption to 9 km/l for every future version object and stop when we find a version object with the consumption in a touched state (for example, the 1970 car has a consumption attribute with a touched flag, so we stop after having propagated the value to the 1950 car). Note that when propagating the consumption to future versions, we do not mark the attribute as touched. It is simply a copy of a predecessor. We only mark an attribute as touched when the user modifies an attribute explicitly. This way, there's a distinction between a user setting a value and the system propagating a value.

A new feature that developers asked for using Temporality is the ability to have attributes that are temporal and others that are not. So, for instance, maybe you want to keep track of the consumption of a car, but you don't want its cost to be. You want the cost property to affect all versions of the car.

To do this, we will add an annotation on the attributes indicating if the attribute is temporal or not.

In order to propagate changes applied to non-temporal attributes to all other versions, we hook into the eSet and check if the attribute is temporal or not. If it is temporal, we do as explained above. However, if it is a non-temporal attribute, we set the value in the current version. All versioned objects look in the current version for non-temporal attribute values.

Here is a bit of pseudo code explaining this behaviour:

```
Temporal.eSet(eObject, feature, value)
        If feature non-temporal
                versionHolder.currentVersion().eSet (feature, value)
        Else
                super.eSet(feature, value)
                setTouchFlag()
       VersionHolder.propagateValue(feature, value, date)

Temporal.eGet(feature)
        If feature temporal
                super.eGet(feature)
        Else
                versionHolder.currentVersion().eGet (feature)

VersionHolder.propagateValue(feature, value, date)
        Starting at temporal version of the given date
           Iterate into the future and set the value (do not set the touch flag) stop when attribute is already
```

touched.

3.4 Hooking Temporality Into EMF

In this section, we will explore various options for hooking temporality into the EMF framework and will explain our decision to hook it into an EStore.

In order for temporality to function properly, it needs to hook into every operation that saves and retrieves an object's attributes. Basically, it needs hooks into the getters and setters of a modeled object.

There are 4 places where it is possible to hook temporality:

- The EMF Notifications
- The generated code
- The EStore
- A base class derived from EStoreEObjectImpl, from which all modeled objects derive.

First, of we can eliminate the notifications option easily since they are only good for setters. They are not supported for getters.

The second option is good since it is very explicit. However, it does not work with dynamically created EClasses.

The EStore option is a good one since it will not only trap every getters and setters but also all access to EList.

The base class option is good because it traps all getters and setters and is fairly explicit.

The last two options are good ones. We will take the EStore option since this is the option taken by the Resource FES as well.

Chances are that in both options we will want to specify our own base class. For the EStore option, this base class would assign the desired EStore. So, we have researched how to specify a custom base class for the EMF objects. It turns out that it is relatively easy to change the base class used by the generated code. All you have to do is change a property in the GenModel GUI.

4 Remaining issues to address

How can we delete a temporal object and still be able to access the old versions?

How to implement temporal history of the car.owner.name attribute and how to persist such states? Perhaps instead of being this sophisticated we could simply say a temporal contained by another temporal may notify his container. Could be an annotation on the property of the contained temporal. When it is modified it triggers a version in the container.

5 Conclusion

We think that this Temporality design is fairly straightforward. It is mostly implemented using modeled EClasses. It does not require any changes to the JET templates used by the GenModel code generator. It does not require any changes to the EMF source code. The Temporality feature is hooked in using an EStore. So, the only code written outside the modeled EClasses is a custom EStore processor that intercepts the eGet and eSet method calls.

As mentioned in the introduction, there are two types of Temporality: Regular and Bi-Temporality. If we ever need to hook in Bi-Temporality, it will be relatively easy. All we would need to do is create a BiTemporal and BiTemporalEList.