Modelling museum context in CIDOC CRM using relational databases

Authors:
Jan Svensson, Anneli Edman, Fredrik Bengtsson, Johanna Zelin, Fredrik Gröndahl

CIDOC06
GOTHENBURG
SWEDEN

Jan Svensson[1], Anneli Edman[2], Fredrik Bengtsson[3], Johanna Zelin[4], Fredrik Gröndahl[5]

1 Introduction

The museums comprise a rich source of knowledge about our cultural heritage. Earlier this knowledge was mainly inherent in professionals at the museums and information was stored in written archives. During the last few decades there has been a transition towards storing the information digitally in relational databases. But storing information is not enough. It is vital to also reproduce peoples' knowledge about the cultural heritage, in a contextual way, for the future. In Sweden there is no general standard for the organization of neither information nor knowledge.

At the moment the museums' databases are isolated from each other and cannot be accessible as a common resource. In the project KMM, Knowledge Management in Museums, we aim to make museums' knowledge accessible in a pedagogical way utilizing IT. We co-operate with five museums, two universities, and commercial partners. To reach the goal we need to store knowledge and information in a structured way. A demand is that the origin of the knowledge is traceable. This leads to a possibility to search for information regarding, e.g., an object and find information related to this based on several different databases. Then we have the base for developing knowledge management systems that allow us to present and utilize the cultural heritage knowledge in a pedagogical context. Our view is that a pedagogical context comprises domain context, pedagogical context, and user context (Edman & Bengtsson, 2006).

[1] Software architect, INYAN AB, Sylveniusgatan 3, SE-754 50 Uppsala, jan.svensson@inyan.se
[2] PhD, Senior lecturer, Department of Information Science, Uppsala University, Kyrkogårdsgatan 10, Box 513, SE-751 20 Uppsala, anneli.edman@dis.uu.se
[3] Master of Computer Science, Research Assistant, Department of Information Science, Uppsala University, fredrik.bengtsson@dis.uu.se
[4] Master of Computer Science, Research Assistant, Department of Information Science, Uppsala University, johanna.zelin@dis.uu.se
[5] Master of Computer Science, Research Assistant, Department of Information Science, Uppsala University, fredrik.grondahl@dis.uu.se

We will use the CIDOC Conceptual Reference Model, CIDOC CRM, for describing the knowledge regarding cultural heritage. The model "*provides definitions and a formal structure for describing the implicit and explicit concepts and relationships used in cultural heritage documentation*"[6]. Having a model is fine, but if it should be utilized in an IT system it has to be implemented. This implementation can, of course, be done in several ways. An important task is to choose how to store the knowledge, e.g. in relational databases, and in that case how these databases should be structured.

First in this article we give the reasons for using a standard, such as the CIDOC CRM model, for the reproducing of cultural heritage. Then we shortly describe the CIDOC CRM model through a small example. After this introduction to the model a way of implementing it in a relational database is described and discussed.

2 Modelling context with CIDOC CRM

Context can be defined as the set of facts or circumstances that surround a situation or event[7]. The situation may include the creators, their purposes, activities and circumstances that caused events to occur[8]. The same reasoning is applicable for information. This means, when reproducing context in a system the model must be able to describe the facts and the circumstances related to these. Moreover, it is vital that the model can be used as a base for the system's reasoning. With reasoning we mean searching and retrieving information and also logical problem solving. Logic is a tool to analyse the relationship between assumptions and conclusions. "*If a conclusion is implied by true or otherwise acceptable assumptions, then logic leads us to accept the conclusion*" (Kowalski, 1979). First order logic comprises the possibility to define *predicates* related to *variables*, and combine these with the connectives *and*, *or*, *implication*, *equivalence,* and *negation*. Moreover, there are *variable quantifiers* referring to their existence in the current domain.

[6] http://cidoc.ics.forth.gr/
[7] wordnet.princeton.edu/perl/webwn
[8] john.curtin.edu.au/society/glossary/

To make the project's museums' knowledge accessible we need to transform the storage of museum information into a standardized format. The International Council of Museums, ICOM, emphasizes the need for standards regarding cultural heritage information[9]. We have chosen the CIDOC CRM standard. The CIDOC CRM is a model expressible in terms of logic or some other suitable knowledge representational form[10]. Sowa argues that knowledge cannot be formalized in logic "*cannot be represented or manipulated on any digital computer in any other notation*" (2000). But logic does not have a vocabulary to describe facts, descriptions or relations, i.e. the predicates. Ontology, though, offers an ordered way of describing these predicates (ibid.). Since the CIDOC CRM is a formal ontology it gives us the means to model the predicates needed to describe the context in a logical system.

In *Figure 1* we see an example describing predicates regarding the relationships between family members. Some of the important terms to understand CIDOC CRM refer to classes, properties and inheritance. Classes are in the CIDOC CRM referred to by an *E* followed by a number and the class name, for example *E67 Birth*. The properties are referred to with a *P* followed by a number and the property name, for example *P4 has time-span*. A class is connected to other classes by properties. The class *Birth* is an event defining the birth of a human being and as such has relations to several persons. Each person is described as an instance of the class *Person*, which in turn is a subclass of the class *Biological Object*. The relation *has time-span*, between *Birth* and the class *Time-Span,* is inherited by *Birth* from a superclass called *Temporal Entity*. In the CIDOC CRM model strict and multiple inheritance are used. Both classes and properties can be inherited.

[9] http://www.willpowerinfo.myby.co.uk/cidoc/stand1.htm#word
[10] Definition of the CIDOC Conceptual Reference Model, Version 4.2, June 2005.

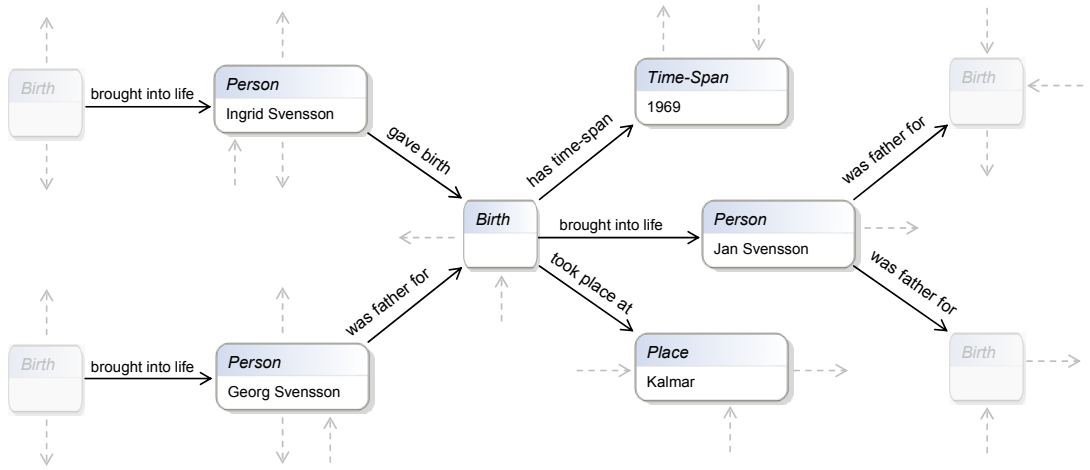

Figure 1. Relationships between family members modelled in CIDOC CRM.

As seen in *Figure 2*, inheritance can span over several classes. *Person* inherits properties from *Biological object*, which defines all objects derived from living organisms. In turn, *Biological object* inherits from the class *Physical object*. This means that the class *Person* has all the properties of its superclasses. Every level, from bottom up, shows a more generalized presentation of the context. This means that the top level is the most abstract one. The top level is a generalized model of the presence of two physical objects at the same event. For each level a narrower specification of the event and objects are made. The consequence is that when modelling a context you describe the most detailed level obtainable and this will not compromise your model.

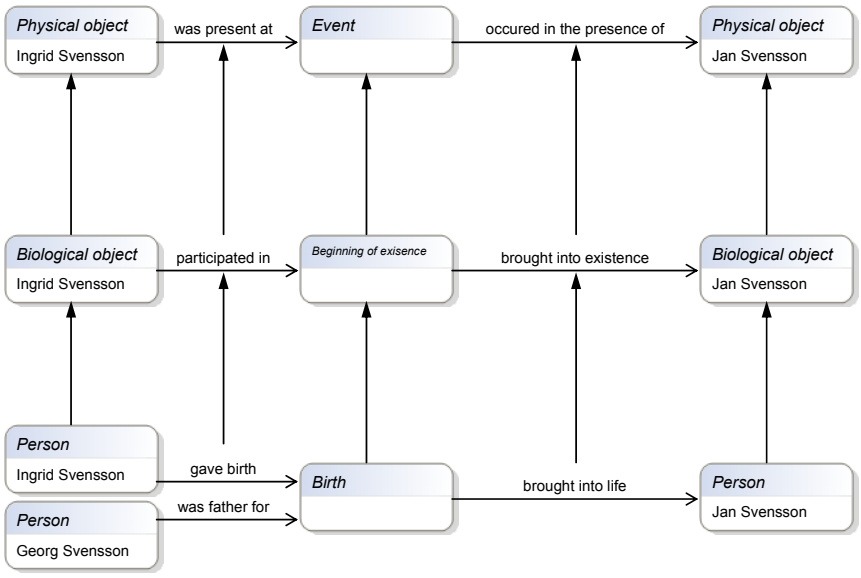

Figure 2. An example of a class hierarchy.

3 Representing CIDOC CRM in a database

In section 2 we discussed why we need a model for dealing with the contextual knowledge within cultural heritage. In this section we explore a possible solution for a database design to structure the information needed for the knowledge representation.

3.1 The importance of inheritance

CIDOC CRM has a comprehensive and powerful object hierarchy, which offers a wide range of abstraction levels through inheritance and it is important to enforce this potential in the database as well. Furthermore, when designing a database that should be able to store everything that can be described with CIDOC CRM, query efficiency becomes a crucial issue.

When one wants to find all events that took place at a certain location, in a database designed according to CIDOC CRM, it is probably not just the instances of the *Event* entity *E5* that one expects to get. One would probably expect to find events like the *Birth* of a person or the *Destruction* of a building. This would, in some sense, make the query object oriented since the query really was for all types of the *Event* entity *E5* and all its descendants.

Any type used in a query will henceforth be referred as the queried type, which always includes all its descendants. As shown in the coming sections, this can be challenging to accomplish in a database but unless it is resolved it would make object inheritance an obstacle instead of being an advantage.

E1 – Entity is the root type for most entities and it implements the property *P2 – has type*, among a few others. Every property is implemented in only one entity type, called its domain type, and the only reason a property exists in more than one entity, is the inheritance between them. The same goes for the other end of the property that only binds to one entity type, called its range type. All properties bind its domain entity to its range entity.

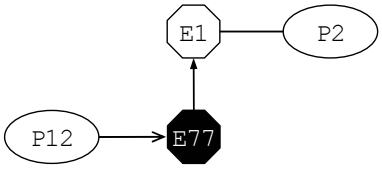

Figure 3

E77 – Persistent Item inherits from *Entity (E1)*, making the property *has type (P2)* implemented for a *Persistent Item (E77)* as well, even though it's not explicitly visualized in *Figure 3*. The *Persistent Item (E77)* is the range type of a few properties, as the illustrated *P12 – was present at*. But these are not valid from *Entity (E1)* since inheritance only goes one way.

E39 – Actor inherits from *Persistent Item (E77)*, which means an *Actor* implements all the properties of *Persistent Item (E77)*. *Figure 4* shows a simplified *Actor (E39)* that is the range type for *P52 – is current owner of* and can be for *was present at (P12)*.

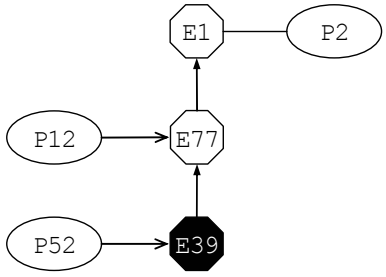

Figure 4

Figure 5 describes the property *is current owner of (P52)* and its range type, an *Actor (E39)*. This means that it links to an *Actor (E39)* or any descendents. They all have the ability to be on the range side of the property, and this is important to remember as shown below.

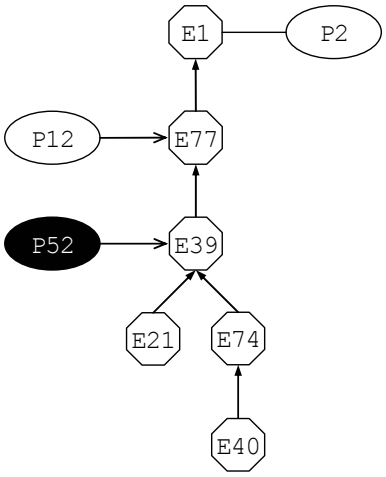

Figure 5

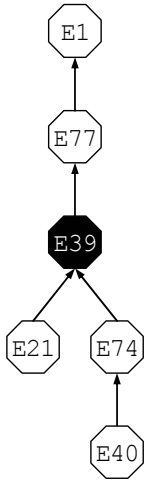

Figure 6

Figure 6 shows the full inheritance hierarchy of an *Actor (E39)* from top to bottom, with its supertypes and subtypes. If we use this approach on *Entity (E1)* that is a root type we get an inheritance hierarchy like in *Figure 7*. The hierarchy of the *Actor (E39)* can be seen in the same figure, going straight down from the top *Entity (E1)*.

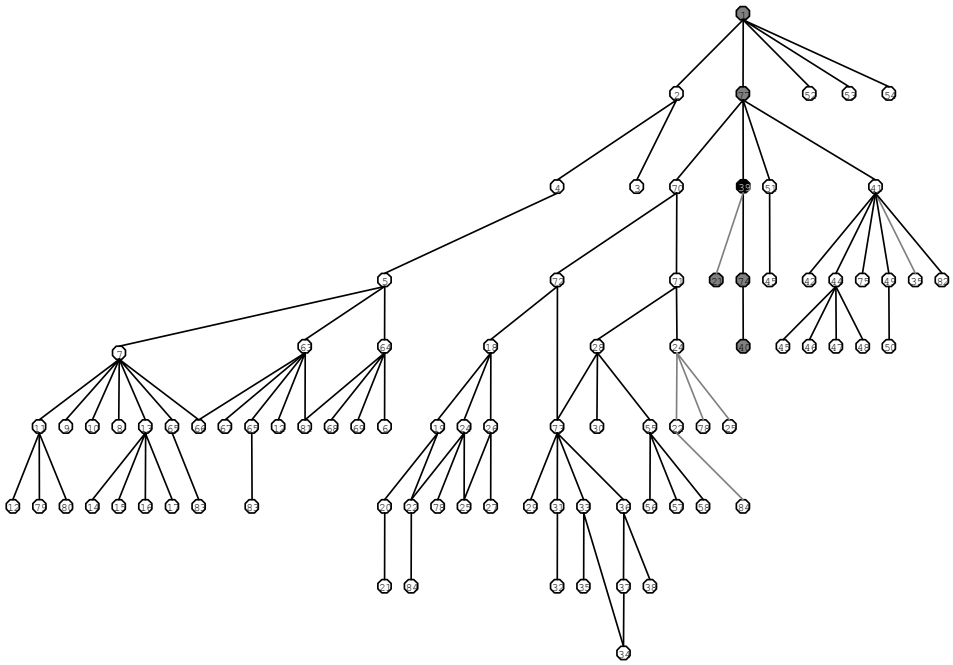

Figure 7

3.2 The proposed database design

Figure 8 shows a data retrieval example scenario:

We want to find every *Actor (E39)* who *is current owner of (P52)* any *Physical Stuff (E18)* that *has type (P2) @GivenTypeID* and that *Physical Stuff (E18)* also *was present at (P12)* any *Event (E5)* that *took place at (P7) @GivenPlaceID*.

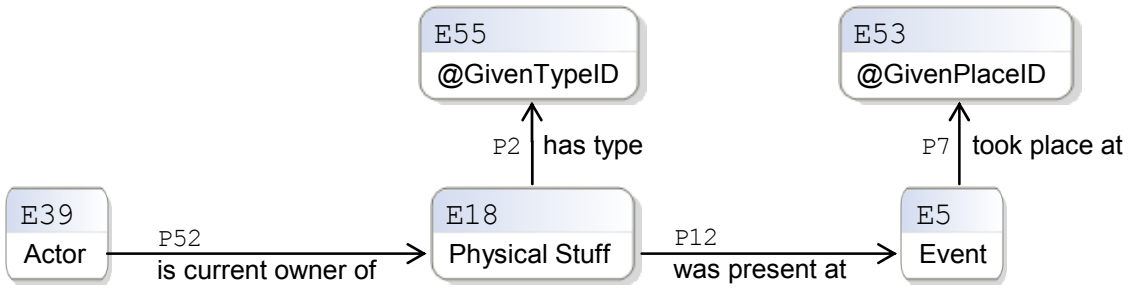

Figure 8

The scenario has been kept small, so that examples to follow will not expand out of proportion. Two variables that have valid IDs are used; this further ensures that the example stays manageable. Thus, we have to imagine that the *@GivenTypeID*, e.g., represents a sword, and the *@GivenPlaceID* can be represented by, e.g., Uppsala.

Figure 9 shows the query scene used for the example scenario. The filled objects are the query types showed in their inheritance hierarchy. Every query type is shown in its full range from top to bottom. The entities *E55 – Type* and *E53 – Place* have been left out since they are not needed. This figure gives some perspective on the query and makes it obvious that the query does not just involve the queried types. All subtypes are involved as well, as discussed for *Figure 5*.

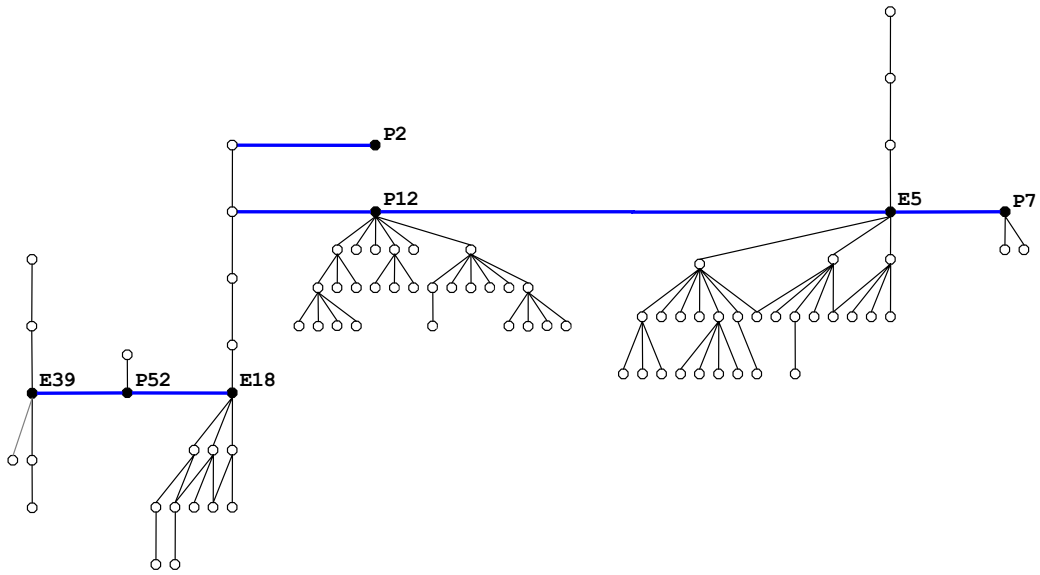

Figure 9

The horizontal lines are the queried properties binding at their domain or range levels. The property *P52 – is current owner of* binds *Actors (E39)* with *E18 – Physical Stuff*. The entity *Physical Stuff (E18)* has two properties that are linked higher up in its hierarchy. Since the queried type is *Physical Stuff (E18)* and it inherits all types higher up in its hierarchy, the query scene in *Figure 10* must be equally true.

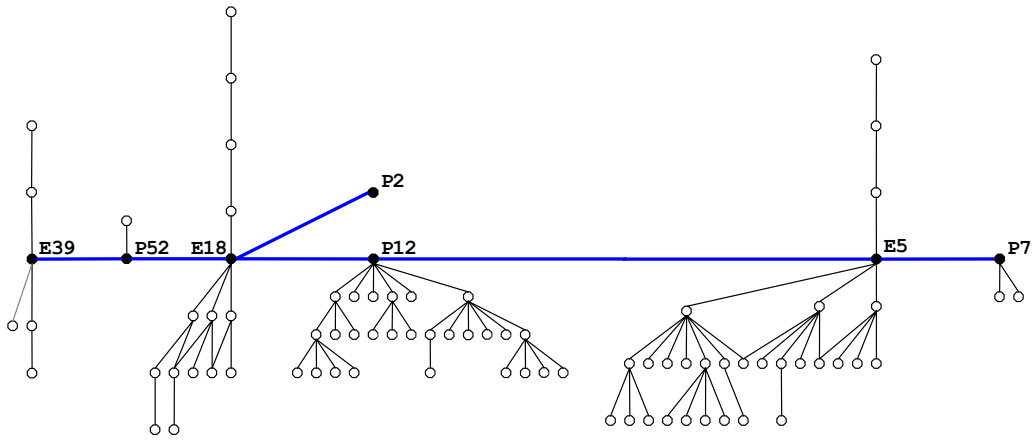

Figure 10

Now we have a query baseline that seems rather straightforward if it wasn't for the subtypes. Both *P12 – was present at* and *E5 - Event* have a lot of subtypes. For instance, the *Event (E5)* query type represents itself and all its subtypes, which means the property *was present at (P12)* not only binds to the *Event (E5)* type but to every subtype of it as well.

Within the stored data, the query type *Event (E5)* could actually be an *E67 – Birth*, the query type *was present at (P12)* could be a *P98 – was born* and the query type *Physical Stuff (E18)* could be an *E21 – Person*. An example of these relations can be seen in Figure 1 and 2. This would make it the birth of a person, which qualifies within "any Physical Stuff that was present at any Event".

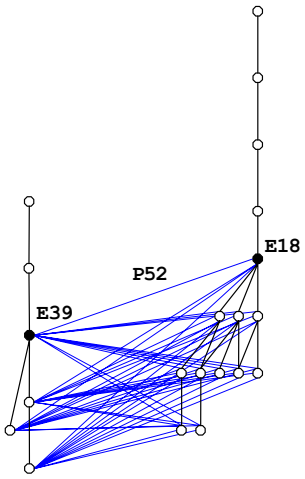

Figure 11

Since the query type *Actor (E39)* represents all types that are an *Actor (E39)* and the query type *Physical Stuff (E18)* represents all types that are *Physical Stuff (E18)*, it's necessary for the query to join all combinations of them, as shown in Figure 11.

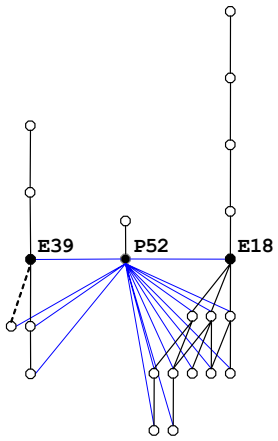

Figure 12

Even though the property *P52 – is current owner of* hides some of the complexity as shown in *Figure 12*, the fact remains that all types of an *Actor (E39)* must join to all types of *Physical Stuff (E18)*.

Figure 13 demonstrates the next step that turns this into an obvious problem. The figure only visualises joins for the queried type and two of its subtypes, leaving the rest of the joins between all *Physical Stuff (E18)* to all types of *P12 – was present at* up to imagination.

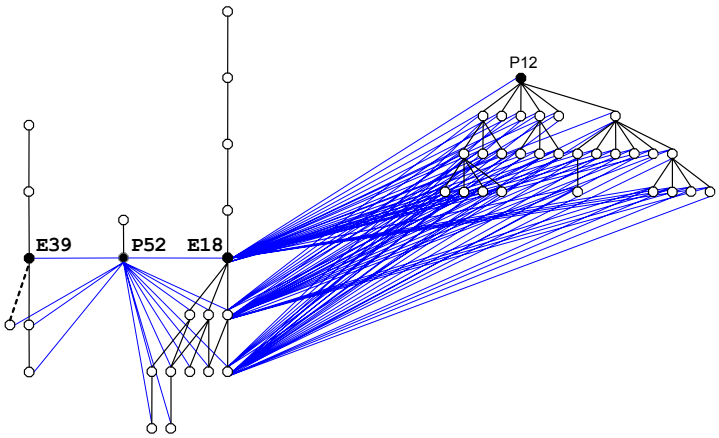

Figure 13

The problem is that these joins can not be handled very well in a query since it would be more than one thousand joins for this simple example scenario.

To be able to execute a reasonably simple query, all subtypes of a query type must be handled as the query type. If this would be done for all query types included in the query, it would only have to join over the path for the query. All subtypes of each queried type would be used as one big

query type, for instance all rows in the database that are subtypes of *Physical Stuff (E18)* would be used as *Physical Stuff (E18)* including *Physical Stuff (E18)* itself. This is illustrated in Figure 14.

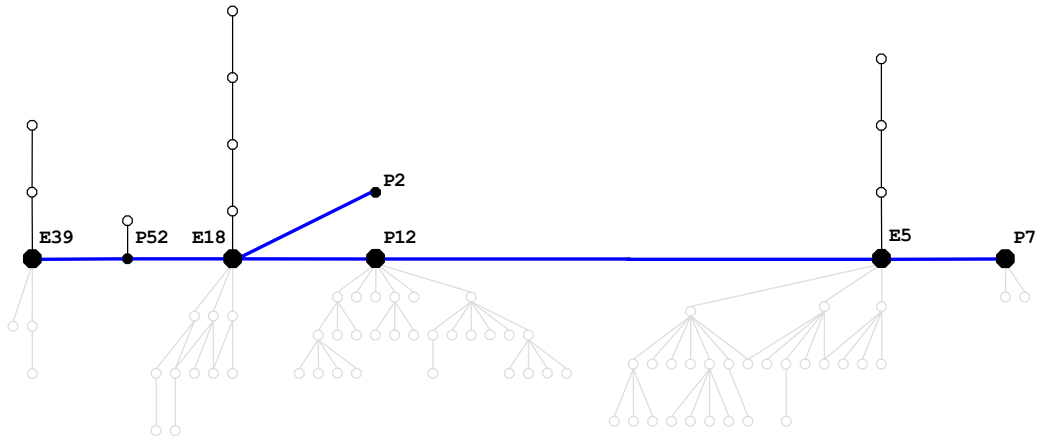

Figure 14

Now the query is straightforward and simple. It would only include one join for each binding, which in this case would be six joins.

The key to this simple query is to handle all subtypes of the queried types as the queried types themselves. If it wasn't for the gathering of subtypes we could probably have typed this query by hand. Depending on database design this gathering would have to be handled either by sub queries, by using aliases, or by unions.

The main reason why this must happen for each query is that we never know what instances will be queried. Every queried type must, each time it is executed, gather all subtypes, and that means extra work for the database engine.

The problem is that we are just storing each type in its current state. For instance, when we store an *Actor (E39)* we would usually only store it as an *E39 – Actor*. However, this does not represent all information regarding *Actor (E39)*. An *Actor (E39)* is basically a *Persistent Item (E77)* with some extra features and so on. If an *Actor (E39)* is stored in the database it should also be stored

as a *Persistent Item (E77)* and as an *Entity (E1)*, because that is the true nature of an *Actor (E39)*. This is the key to how we can make inheritance work to our advantage, simply by storing all types in all their inherited types as well. Since all types are stored in all their instances they can always be retrieved from any instance. This really means that we don't have to gather any subtypes of the queried types, simply because they already exist as the queried type. Even though the quantity of rows would be exactly the same, the rows would not have to be gathered from a range of other types. The query for Figure 14 would now be as easy as any traditional query in a relational database and could definitely be typed by hand if needed.

The database model is implemented as a relational database with as many tables as types. Each table is constrained to ensure the inheritance hierarchy and for type safety of properties. Stored procedures are implemented to manage data storage and modification.

4 Conclusion and discussion

CIDOC CRM has a comprehensive and powerful object hierarchy, which offers a great level of abstraction through inheritance and it is important to enforce this potential in the database. To fully utilize the strength in a class hierarchical model the database model also should reflect the inheritance aspect. We argue that to be able to retrieve information in such a database we need to store all types in all their inherited types as well.

Important is, as we see it, that the CIDOC CRM model is built around conceptual modelling. Thus, we are able to mirror the conceptual model in our database design. This gives us the possibility to model the knowledge without being restricted by the database model, since the conceptual model and the database model share the same structure. Furthermore, this minimizes the need for translation between the theoretical model and its representation in the database. A major advantage is that the database itself functions as a schema for validating the theoretical model when data is stored in the database. For instance, it is not possible to store the wrong properties associated with a specific class. Moreover, the CIDOC CRM model is founded on a

logical base, which in turn means that our proposed database design also has the same logical properties.

Some tests have been performed and these show that the performance when querying is satisfying. Further testing of the proposed database solution is needed, both of real case scenarios already mapped into CIDOC CRM and through mathematical methods. Moreover, we need to test how efficient it is to model the contextual knowledge in the database. Further work also includes investigating methods for the implementation of the user interaction with the model. An important issue is to study different approaches to search and retrieve information. This issue is complicated. Firstly, the demand regarding queries is multifaceted due to the complicated structure of the database. Secondly, to utilize the possibility to retrieve contextual knowledge we need to find new ways to extract it.

We have discussed the possibilities when using the CIDOC CRM model to structure cultural heritage information to be used in a contextual manner. Furthermore, a database design that could be a candidate for hosting this information is proposed. It is vital, though, to acknowledge that this is a part of a larger structure, i.e., the knowledge management in museums. A process pattern for this can comprise capturing, storing, sharing, and utilizing knowledge (Edman, 2006).

References

CIDOC Conceptual Reference Model, Version 4.2, June 2005. ICOM/CIDOC CRM Special Interest Group.

Edman, A. (2006) A process pattern for knowledge management in museums. Proceedings of CIDOC06, Gothenburg, Sweden.

Edman, A. & Bengtsson, F. (2006) Museum context in a pedagogical environment. Proceedings of CIDOC06, Gothenburg, Sweden.

Kowalski, R. (1979) Logic for problem solving. Elsevier Science Publishing. North-Holland.

Sowa, J.F. (2000) Knowledge representation - Logical, philosophical, and computational foundations. Thomson Learning. Brooks/Cole.