Towards the Mapping of Dynamic Properties of Objects, described by UML State Diagrams, on Relational Databases

José M. Maciel
University of Caxias do Sul, Social Sciences and Communication Dept.,
Vacaria – RS, Brazil, 95200-000
jmmaciel@ucs.br

and

Duncan D. Ruiz
Pontifical Catholic University of RS, Dept. of Applied Computing,
Porto Alegre – RS, Brazil, 90619-900
duncan@inf.pucrs.br

Abstract

The work presents an approach to map objects behavior, described by UML State Diagrams, to Relational Database Systems description models by the use of SQL Triggers (E-C-A rules of active databases). The UML State Diagram concepts properly handled by our approach are: **states, composite states** (without concurrent sub-states), **events, simple transitions** and **transitions to and from composite states**, according with the terminology presented by the latest OMG-UML standard (1.4). We argue the intra-objects dynamic properties can be properly implemented using triggers, by the use of implementation patterns and well-defined mapping rules. The contribution of this research is a method to describe a larger part of an application being modeled into the application database and managed by the corresponding DBMS. This approach relieves the amount of programming needed and improves the autonomy of such application database. In addition, such approach improves the robustness of databases against undesirable changes.

Keywords: Relational databases, Active databases, SQL Triggers, UML**.**

1. Introduction

One of the present challenges in computer science is to reduce costs and improve the quality of the solutions based on computer systems – solutions that are basically implemented by computer software [1]. The object-oriented paradigm applied to software engineering intends to satisfy the needs of improving the quality for the development and maintenance of computer software and to succeed the structured methods of analysis and design [2] [3]. After few years, the Unified Modeling Language - UML emerged as the dominant language in the software industry and has been adopted as an object-modeling standard by OMG [4]. UML offers a set of integrated diagrammatic tools permitting the representation and specification of information systems [5]. In this sense, it is observed a growing use of the object-oriented paradigm, and hence of the UML, during the modeling process of systems with data persistence. To obtain such persistence, it is recommended the use of some DBMS technology.

Today the relational database approach is currently the "de facto" standard for the management of data by small and large companies. This situation occurs due to several reasons: (1) legacy databases need to remain available [6], and (2) it is a well-tested and reliable technology, for a long time, and it is well understood. Mapping objects to relational database description models is a problem that requires better solutions because the software designers use object-oriented models during the software analysis and project design, and implement such software over relational databases. There are several conceptual differences between object-oriented paradigm and the relational model.

Due to the growing use of UML to model information systems, it is important the development of mapping methods from UML models to database implementation models, especially with respect to the mapping of dynamic properties described by state diagrams. We believe that the mapping of these dynamic properties into the database description model will permit that such database becomes selective with respect to changes of their data, accepting only that changes previously modeled and, consequently, considered suitable. The mapping of dynamic aspects into the description model of the DBMS is a valuable feature to be accomplished during the database design because a larger part of the application semantics will be supported by the DBMS. As a consequence, the implementation of the application programs may be alleviated. Other advantages of this approach are: (1) speeding up the building of applications based on databases, (2) better managing of the integrity constraints of the attributes, and (3) better representation of the application real world into its programs.

Several authors have devoted their time to describe how to translate objects to the relational description model. They have proposed different ways to perform this mapping and, recently, some of them are using pattern languages. Almost all approaches are centered on mapping the static properties of objects [7] [8] [9] [10] [11] [12]. Few authors address the mapping of some dynamic properties. [13] proposes a mapping model to object-oriented programming languages. [14] describes a generic pattern language to map state diagrams to an object-oriented language. [15] discusses the mapping of static properties of UML classes using a new normal form called Nested Normal Form [16], and addresses some desirable aspects to be satisfied by a set of triggers that implement dynamic properties described by UML state diagrams.

In this paper we present a method to help application designers, which uses UML state diagrams to model dynamic properties of applications, in the task of mapping such properties automatically into the relational description model. As we will see ahead, our approach deals with **states** and **composite states** without concurrent sub-states, **events**, **simple transitions** and **transitions to and from composite states**. To make it feasible, we are considering the functionality offered by SQL-triggers, conforming to the current SQL/ANSI standard: SQL:1999 [17]. To validate our approach we used the DBMS Oracle, which almost conforms to such SQL standard.

The paper is organized as follows. Section 2 briefly presents the basic notation of UML state diagrams and the case study used to demonstrate the quality of our approach. The method proposed is presented in Section 3. Section 4 presents the related work concerning the implementation of classes and state diagrams and Section 5 presents the conclusions and future work.

2. UML State Diagrams

In this section it is presented the basic notation for UML state diagrams as documented in [18] and [19]. The goal is to briefly describe the UML state diagram concepts targeted by our approach. Also, it is described a case study that will be used in Section 3 to document the application of the proposed method. Such case shows an UML class and its state diagram to demonstrate the effectiveness of our approach. The case is the document digital publishing of a university [23,24].

UML state diagrams intend to represent the entity behavior through a sequence of states that a particular object can be during its life cycle. The notation and semantics of state diagrams is based on Statecharts [20][21], and Statecharts are based on Automata theory [22]. According with [18], UML state diagrams presents the concepts of **states** and **composite states** (with or without concurrent sub-states), **events**, **actions, activities, guard conditions**,

invocation of nested state machines, **simple transitions**, **transitions to and from concurrent states, transitions to and from composite states, factored transition paths, submachine states, synch states** and **history state indicator**. Figure 1 depicts an UML state machine, similar of the presented by [19]. Such state machine was used by [19] to show the main concepts of the UML graphical notation.

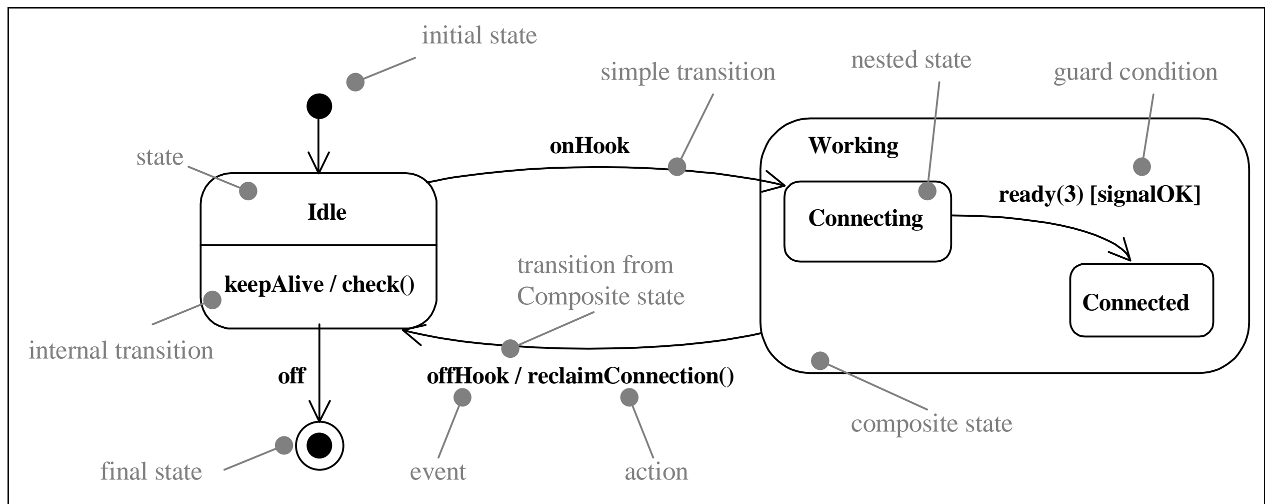

Figure 1: Main concepts of UML State Diagrams.

We do not address the mapping of UML state diagram concepts that need some level of programming. **Actions** and **activities** imply the development of functions on some programming language. Also, Boolean functions can be used on **guard conditions**. Similarly, **history state indicator** needs some new attributes to be included in the class specification. In spite of the syntax of SQL triggers offered by the main relational SGBDs are almost conform the SQL:1999 standard, we do not address **concurrent states** because such SGBDs differ on the ways concurrent triggers are handled. These concepts mentioned above plus the advanced ones presented in the latest version of OMG-UML standard (**factored transition paths, submachine states, synch states**) will be addressed in our future work. All other concepts are used in their standard meanings.

2.1 A case study: document digital publishing system

The goal of this system is to monitor the digitalizing, formatting and indexing process of documents, e.g., PhD thesis and MSc dissertations, research reports, technical manuals, etc. *Document* is the main class of such system for digital publishing by a digital library [23] [24]. Figure 2 shows the class *Document* and its state diagram for a document preparation system.

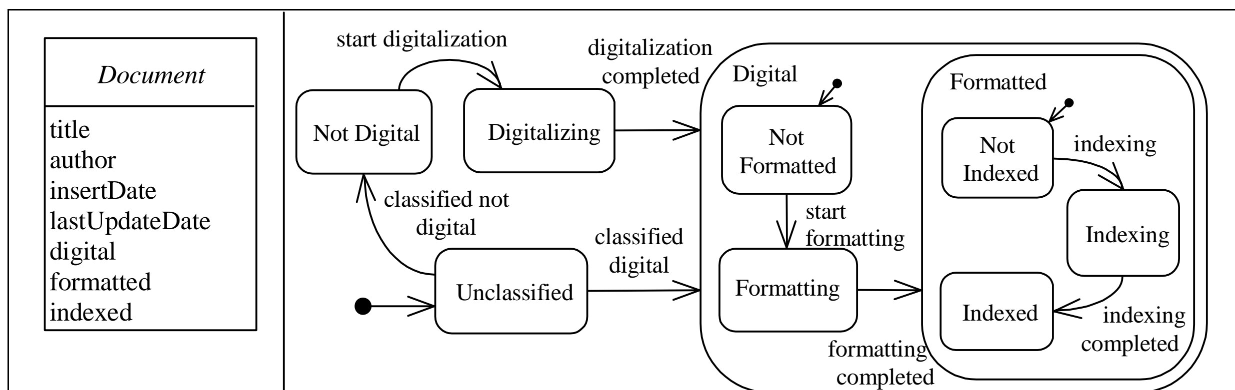

Figure 2 UML Class *Document* and corresponding UML state diagram.

Document dynamic properties description:

The preparation activities start at receiving one document to be included into the collection of the Digital Library. This document may be received by two different ways: Not Digital (paper document) or Digital (computer file). After its classification, in case of a not digital document, it is necessary to scan it and generate its corresponding

digital file (.PS, .PDF, .HTML, etc.). With the digital version of the document, it is formatted according with the rules and standards defined by the Digital Library. The document formatting stage will permit the access of any document by the same interface type, e.g., HTML, XML, etc. Finally, the document is indexed by the creation of entries into the several access structures of the system, typically indexes by authors, subjects, keywords, years of publication, etc. After these stages, the document will be available to be retrieved and accessed.

The digitalizing process may spend from few minutes to various days, according several factors such as size and complexity of the document. To be considered ready to be electronic published, an object must follow several stages, modeled as states, according with the state diagram showed in that is detailed above. When the document become ready for electronic publishing, it is moved to the digital library system, whose functionality will not be discussed here due to the lack of space. More details may be found at [25].

Figure 2 depicts the following UML state diagram concepts: **state, composite state, event, simple transition** and **transition to composite state.** We have chosen the class *Document* of this system because document objects have interesting behavior and present several possible configurations into its attributes that characterize different states in which these objects must follow during the preparation process. The class Document is implemented in SQL using simple attributes to better demonstrates the process mapping.

3. The mapping process

To map UML classes and its dynamic aspects modeled by state diagrams, our mapping process is divided into four steps to be applied on each UML class with state diagrams.

1. The mapping of the static properties to relational tables.
2. The definition of the semantics for each object state, presented in its state diagram, and the insertion into a Dictionary of states.
3. The definition of the semantics of each valid transition, among object states, and the insertion into a Dictionary of valid transitions.
4. The creation of database triggers based on the dictionaries built at previous steps.

3.1 Mapping of the static properties

The first step of our mapping process is the mapping of static properties to relational tables. Such mapping of each UML class to the relational description model is done according to the approach presented in [25], which is a well-known and well-accepted approach. Due to the lack of space, we will concentrate the explanation of the static mapping considering only UML classes with atomic attributes and with simple domains. In special, we considered that the object identity is mapped by inserting an object identifier – OID (it is chosen `char(5)` as OID domain only for explanation purposes). As a result, for each UML class corresponds to one table into the relational database. Considering the class *Document* presented in Figure 2, the resulting relation may have the following description, in Oracle SQL:

```
create table Document(  OID   char(5)  not null,   author        char(30) not null,
                        title char(30) not null,   insertDate    date     not null,
                        lastupdateDate  date,      digital         char(1),
                        formatted       char(1),   indexed         char(1),
primary key (OID));
```

3.2 Definition of the semantics of each object state

The second step of our mapping process is the definition of the semantics for each object state. In this step the semantics of each object state must be defined in terms of its attribute values. It is considered that all object states can be defined considering the combinations of its attribute values. The resulting semantics is stored in a Dictionary of states. Each entry of this dictionary is a 4-uple (Class name, State, Parent state, Semantics). **Class name** is the name of the class being mapped. **State** is a valid state name of **Class name. Parent state**, if present, is the super-state of **State**. **Semantics** is the definition of the **State** in terms of **Class name** attribute values. If **State** has **Parent state**s, its actual semantics is the combination of its proper **Semantics** with the **Semantics** of all its parent states.

The Dictionary of states, for each UML class, must satisfy the following consistency rules.

- All states defined in the state diagram must be presented.
- One state must not be presented more than once.

- The semantics of each elementary state, defined according possible combination of values of the object attributes, must be mutually exclusive regarding the semantics of all other elementary states.

For our case study, the semantics of each *Document* state is stored in the Dictionary of states in the way presented by Table 1. For example, the actual semantics of state Indexing is “Digital = 'Y' and Formatted = 'Y' and Indexed = 'I'”.

Table 1- Dictionary of states - semantics of each *Document* state.

Class name	State	Parent State	Semantics
Document	Unclassified		digital is null and formatted is null and indexed is null
Document	Not Digital		digital = 'N' and formatted is null and indexed is null
Document	Digitalizing		digital = 'D' and formatted is null and indexed is null
Document	Digital		digital = 'Y'
Document	Not Formatted	Digital	formatted = 'N' and indexed is null
Document	Formatting	Digital	formatted = 'F' and indexed is null
Document	Formatted	Digital	formatted = 'Y'
Document	Not Indexed	Formatted	indexed = 'N'
Document	Indexing	Formatted	indexed = 'I'
Document	Indexed	Formatted	indexed = 'Y'

3.3 Definition of the semantics of each valid transition

In this step, it is identified the semantics of each transition considered valid to occur in the state diagram of an UML class. In our approach, it is considered that each transition is a modification operation, e.g., INSERT, UPDATE or DELETE, on an object of the class. A correct transformation of attribute values of an object must correspond to one valid event in its state diagram. The creation of an object, e.g., the transition from <initial state> to the first valid state, is an INSERT operation. The transition to one <final state>, if present, is a DELETE operation. We have chosen to consider the <final state> as a deletion of the object, instead of consider it as a frozen state, to permit the real exclusion of the object from the database. If there is a real situation where it must not be done, it is enough that the state diagram of the corresponding UML class does not have any <final state>. All other transitions among states, excluding <initial state> and <final state>s are UPDATE operations. The resulting semantics is stored in a Dictionary of valid transitions. Each entry of this dictionary is a 5-uple (**Class name, Event, From State, To State, Operation**). **Class name** is the name of the class being mapped. **Event** is the name of the event, if present. **From State** and **To State** are valid **State**s of the class and record the start and the end of each transition arrow. **Operation** is the modification operation to be monitored by the trigger.

The dictionary of valid transitions must satisfy the following consistency rules.

- All valid events of the state diagram must be presented. <initial state> must occur only once and in the column **From State**.
- <final state> may occur zero or more times, every in the **To State** column.
- The states that occur in the **To State** column can’t have sub-states. In state diagrams without concurrent sub-states, it is always possible to find an elementary state as the destination state for any event.
- The **Operation** in the row with <initial state> is INSERT compulsorily.
- The **Operation** in the rows with <final state>, if present, is DELETE compulsorily.
- The **Operation** in the rows without <initial state> and <final state> is UPDATE.
- All the states listed into the Dictionary of valid transitions must appear into the Dictionary of states, except the <initial state> and <final state> states.

For our case study, the semantics of each *Document* event is stored in the Dictionary of valid transitions in the way presented on Table 2.

Table 2- Dictionary of valid transitions - semantics of each *Document* event.

Class name	Event	From State	To State	Operation
Document	<creation of the object>	<initial state>	Unclassified	INSERT
Document	classified not digital	Unclassified	Not Digital	UPDATE
Document	classified digital	Unclassified	Not Formatted	UPDATE
Document	start digitalization	Not Digital	Digitalizing	UPDATE
Document	digitalization completed	Digitalizing	Not Formatted	UPDATE
Document	start formatting	Not Formatted	Formatting	UPDATE
Document	formatting completed	Formatting	Not Indexed	UPDATE
Document	indexing	Not Indexed	Indexing	UPDATE
Document	indexing completed	Indexing	Indexed	UPDATE

3.4 Creation of database triggers

In this step the database triggers for the relational table resulted from the mapping of an UML class are automatically built considering the contents of both dictionaries – Dictionary of states and Dictionary of valid transitions. These triggers will guarantee that only valid events will be accepted, i.e., only previously defined attribute changes can occur.

3.4.1 General rules for triggers creation

The triggers must be of the type BEFORE because such triggers must reject invalid changes and, hence, it is better that the rejection happens before the physical update of the object. And it is necessary to prefix all the attributes with :old or :new properly to refer the attribute values before or after the modification. The triggers are built with the clause FOR EACH ROW because we are mapping the dynamic properties of one object (one UML class instance). The list of monitored attributes, into the definition of the trigger for UPDATE operations, permits the selective execution of this trigger. Indeed, the changes into the attributes that do not influence on the semantics of the states need not be monitored. In the case of a **State** with a valid **Parent state**, its actual semantics is its own semantics combined with the semantics of all of its parent states.

3.4.2 INSERT triggers

This trigger monitors the creation of objects with acceptable values in its first state. An algorithm able to produce an INSERT trigger, for each **Class name**, is the following.

1. Search the row *r-vt* in the table Dictionary of valid transitions where **From State** equals “<initial state>”.
2. Retrieve *r-vt*.**To State** value in the row found by step 1.
3. Search the row *r*-s in the Dictionary of states where *r-vt.***To State** equals *r-s*.**State.**
4. Create a BEFORE INSERT trigger named *insert<r-vt.***Class name>** on relation *r-vt.***Class name**.
5. In the trigger body, insert an *if statement* where NOT(*r-s*.**Semantics)** is the condition and the attributes present in *r-s*.**Semantics** must be prefixed by “:new.”.
6. Complete properly the trigger definition and insert a command to abort the transaction in case of invalid transition.

The trigger built for the INSERT into the *Document* relation, using the SGBD Oracle, is:

```
create or replace trigger insertDocument
before insert on Document for each row
begin
 if not(:new.digital is null and :new.formatted is null and :new.indexed is null)
 then raise_application_error(-20500,'Invalid State Transition',true);
 end if;
end;
```

3.4.3 DELETE triggers

This trigger monitors the possible deletion of objects. A **Class name** object can only be deleted if there is <final state> in the column **To State** on some row of the table Dictionary of valid transitions regarding **Class name.** In addition, the **Class name** object must be in one state connected to a <final state> by a valid transition. An algorithm able to produce DELETE triggers, for each **Class name**, is the following.

1. Search the rows *r-vt* in the table Dictionary of valid transitions where **To State** equals "<final state>".
2. Create a BEFORE DELETE trigger named *delete*<**Class name**> on relation **Class name**.
3. If exists at least one *r-vt* then:
   3.1. In the trigger body, insert an *if statement.*
   3.2. For each *r-vt* found.
      3.2.1. Retrieve *r-vt*.**From State** value.
      3.2.2. Search the row *r-s* in the Dictionary of states where *r-vt*.**From State** equals *r-s*.**State.**
      3.2.3. Prefix the attributes present in *r-s*.**Semantics** by ":old." and insert NOT(*r-s*.**Semantics)** in the *if condition* of the inserted *if* command by step 3.1.
   3.3. Insert AND operators between each pair of NOT(*r-s*.**Semantics)** inserted by step 3.2.
4. Complete properly the trigger definition and insert a command to abort the transaction in case of invalid transition.

The trigger built for the DELETE from the *Document* relation is:

```
create or replace trigger deleteDocument
before delete on Document for each row
begin
  raise_application_error(-20500,' Invalid State Transition',true);
end;
```

3.4.4 UPDATE triggers

The basic structure for building the UPDATE trigger starts with checking if the UPDATE statement really wants to change the object state. If the answer is yes then such trigger checks if there is a valid transition defined between **From State** and **To State**. An algorithm able to produce UPDATE triggers, for each **Class name**, is the following.

1. Create a BEFORE UPDATE trigger named *update*<**Class name**> on relation **Class name**.
2. Insert all attributes presented in *r-s*.**Semantics** of any *r-s* row of Dictionary of states into the trigger clause UPDATE OF, for *r-s*.**Class name** equals **Class name**.
3. For each attribute found by step 2, insert a condition to check if the UPDATE statement changes its value.
4. Combine properly the conditions inserted by step 3 to determine if it is necessary to check if there is a valid transition between **From State** and **To State**.
5. If it is necessary to check if the transition is valid, then:
   5.1. For each row *r-s* of Dictionary of states where *r-s*.**Class name** equals **Class name** and *r-s*.**Parent State** is not defined.
      5.1.1. Insert an *if statement* in the trigger body where its condition is *r-s*.**Semantics** and prefix all its attributes by ":old.".
      5.1.2. For each row *r-vt* of Dictionary of valid transitions where *r-vt*.**Class name** equals *r-s*.**Class name** and *r-vt*.**From State** equals *r-s*.**State.**
         5.1.2.1. Retrieve row *r-s1* of Dictionary of states where *r-s1*.**Class name** equals *r-s*.**Class name** and *r-s1*.**State** equals *r-vt*.**To State.**
         5.1.2.2. Insert an *if statement*, nested to the inserted by step 5.1.1, where its condition is *r-s1*.**Semantics** and prefix all its attributes by ":new.".

5.1.2.3. If is there some row *r-s2* of Dictionary of states where *r-s2*.**Class name** equals *r-s*.**Class name** and *r-s2*.**Parent State** equals *r-s*.**State,** then apply recursively steps 5.1.1 and 5.1.2 considering *r-s2* the new *r-s*.

6. Complete properly the trigger by connecting *if statements* and inserting commands to abort the transaction in case of an invalid transition.

The trigger built for the UPDATE of *Document* relation is listed in [25]. It is presented below the code lines with additional comments to show how the above method generates the trigger.

- **Trigger header – steps 1 and 2:**

```
create or replace trigger updateDocument
before update of digital, formatted, indexed on Document for each row
```

- **Check if it is necessary to check the transitions – steps 3 and 4:**

```
begin -- checking if it is a real state transition
if (:old.digital is null and :new.digital is not null or :old.digital is not null and
    :new.digital is null or :old.digital is not null and :new.digital is not null and
    :old.digital <> :new.digital)
or (:old.formatted is null and :new.formatted is not null or :old.formatted is not
null
    and :new.formatted is null or :old.formatted is not null and :new.formatted is
    not null and :old.formatted <> :new.formatted)
or (:old.indexed is null and :new.indexed is not null or :old.indexed is not null and
    :new.indexed is null or :old.indexed is not null and :new.indexed is not null and
    :old.indexed <> :new.indexed)
then
```

- **Check the acceptable events for the state unclassified – step 5:**

```
if (:old.Digital is null and :old.Formatted is null and :old.Indexed is null)
 then -- From State UNCLASSIFIED --
  if (:new.Digital='NO' and :new.Formatted is null and :new.Indexed is null )
  then -- Event CLASSIFIED NOT DIGITAL --
  elsif (:new.Digital='YES' and :new.Formatted='NO' and :new.Indexed is null)
  then -- Event CLASSIFIED DIGITAL --
  else raise_application_error(-20501,'Invalid State Transition',true);
  end if;
```

- **Check the acceptable events for the state not digital – step 5:**

```
elsif (:old.Digital='NO' and :old.Formatted is null and :old.Indexed is null)
 then -- From State NOT DIGITAL --
  if (:new.Digital='DIGITALIZING' and :new.Formatted is null and :new.Indexed is null)
  then -- Event START DIGITALIZATION --
   dbms_output.put_line('State Transition START DIGITALIZATION');
  else raise_application_error(-20502,'Invalid State Transition',true);
  end if;
```

- **Check the acceptable events for the state digitalizing – step 5:**

```
elsif (:old.Digital='DIGITALIZING' and :old.Formatted is null and
       :old.Indexed is null)
then -- From State DIGITALIZING --
  if (:new.Digital='YES' and :new.Formatted='NO' and :new.Indexed  is null)
  then -- Event DIGITALIZATION COMPLETED --
   dbms_output.put_line('State Transition DIGITAL and NOT FORMATTED');
  else raise_application_error(-20503,'Invalid State Transition',true);
  end if;
```

- **Check the acceptable events for a composite state digital – step 5:**

```
 elsif (:old.Digital='YES')
 then -- From State DIGITAL –
```

- **Check the acceptable events for a composite state digital / not formatted – step 5 recursively:**

```
  if (:old.Formatted='NO' and :old.Indexed  is null)
  then -- From State DIGITAL and NOT FORMATTED --
   if (:new.Digital='YES' and :new.Formatted='FORMATTING' and :new.Indexed is null)
   then -- Event FORMATTING --
    dbms_output.put_line('State Transition DIGITAL and FORMATTING');
   else raise_application_error(-20504,'Invalid State Transition',true);
   end if;
```

- **Check the acceptable events for a composite state digital / formatting – step 5 recursively:**

```
  elsif (:old.Formatted='FORMATTING' and :old.Indexed is null)
  then -- From State DIGITAL and FORMATTING --
   if (:new.Digital='YES' and :new.Formatted='YES' and :new.Indexed='NO')
   then -- Event FORMATTING COMPLETED --
    dbms_output.put_line('State Transition DIGITAL,FORMATTED and NOT INDEXED');
   else raise_application_error(-20505,'Invalid State Transition',true);
   end if;
```

- **Check the acceptable events for a composite state digital / formatted – step 5 recursively:**

```
  elsif (:old.Formatted = 'YES')
  then -- From State DIGITAL and FORMATTED --
```

- **Check the acceptable events for a composite state digital / formatted / not indexed – step 5 recursively:**

```
   if (:old.Indexed  = 'NO')
   then -- State DIGITAL, FORMATTED and NOT INDEXED --
    if (:new.Digital='YES' and :new.Formatted='YES' and :new.Indexed='INDEXING')
    then -- Event INDEXING --
     dbms_output.put_line('State Transition DIGITAL,FORMATTED and INDEXING');
    else raise_application_error(-20506,'Invalid State Transition',true);
    end if;
```

- **Check the acceptable events for a composite state digital / formatted / indexing – step 5 recursively:**

```
   elsif (:old.Indexed  = 'INDEXING')
   then -- From State DIGITAL, FORMATTED and INDEXING --
    if (:new.Digital='YES' and :new.Formatted='YES' and :new.Indexed='YES')
    then -- Event INDEXING COMPLETED --
     dbms_output.put_line('State Transition DIGITAL,FORMATTED and INDEXED');
```

- **Complete properly the trigger – step 6:**

```
    else raise_application_error(-20507,'Invalid State Transition',true);
    end if;
   else dbms_output.put_line('State Transition DIGITAL,FORMATTED and NOT INDEXED');
   end if;
  else raise_application_error(-20508,'Invalid State Transition',true);
  end if;
 else raise_application_error(-20509,'Invalid State Transition',true);
 end if;
end if;
end;
```

3.5 Experiments

Considering the UML class *Document*, used as a case study, we submitted a set of modifying statements exhausting all combinations of values from the monitored attributes of *Document* and validating all valid/invalid state transitions. The complete set of tests is documented in [25]. Due to the lack of space, we will show some examples applied to the case study. According the sequence presented below, there are: (1) a creation of a document in the valid state UNCLASSIFIED, (2) an attempt to insert a document in an invalid state, (3) a valid state transition for a document, (4) an invalid state transition, and (5) an attempt to delete a document.

```
SQL> insert into Document(OID,author,title,insertDate,lastupdateDate,digital,
  2  formatted,indexed) values ('00001','John Doe',
  3  'Temporal Databases: introd.','13-NOV-2000',Null,Null,Null,Null);
State Transition UNCLASSIFIED
1 row created.

SQL> insert into Document(OID,author,title,insertDate,lastupdateDate,digital,
  2  formatted,indexed) values ('00002','Mary Doe',
  3  'Mapping Objects to SQL','13-NOV-2000', Null,Null,Null,'Y');
insert into Document(OID,author,title,insertDate,lastupdateDate,digital,
*
ERROR at line 1:
ORA-20500: Invalid State Transition
ORA-06512: at "TEST.INSERTDOCUMENT", line 3
ORA-04088: error during execution of trigger 'TEST.INSERTDOCUMENT'

SQL> update Document set digital='N',formatted=Null,indexed=Null
  2  where OID='00001';
State Transition NOT DIGITAL
1 row updated.

SQL> update Document set digital='N',formatted=Null, indexed='N'
  2  where OID='00001';
update Document set digital='N',formatted=Null, indexed='N'
*
ERROR at line 1:
ORA-20502: Invalid State Transition
ORA-06512: at "TEST.UPDATEDOCUMENT", line 27
ORA-04088: error during execution of trigger 'TEST.UPDATEDOCUMENT'

SQL> delete from Document where OID='00001';
delete from Document where OID='00001'
*
ERROR at line 1:
ORA-20500:  Invalid State Transition
ORA-06512: at "TEST.DELETEDOCUMENT", line 2
ORA-04088: error during execution of trigger 'TEST.DELETEDOCUMENT'
```

3.6 Properties of the triggers produced

Considering the properties of **termination, confluence** and **observable determinism**, the triggers produced by the mapping process described in this paper satisfy all of them. This is true because there are produced only 3 triggers, one for each type of modification statement. We opted to generate only one trigger for each type of modification statement because it is known that commercial DBMS presents some difficulties to support efficiently a large number of defined triggers. In addition, the implementation logic does not use iterative, asynchronous or "go to" statements. Then the logic of each trigger is executed in a top-down way having only one thread of execution. However, if the designer implements his/her own triggers, these properties must be properly evaluated.

The limitations of our approach are (1) the partial support of state diagram functionalities, and (2) UML classes that their states can be defined only by attribute values. In fact, **composite states** with concurrent sub-states and **complex transitions** (concurrent threads of control) require asynchronous execution into the trigger logic or parallel execution of triggers. And **history state indicator** requires the record of the past values of the attributes, or a special attribute to store previous states.

4. Related Work

Several authors have devoted their time to describe how to translate objects to the relational description model. Almost all approaches are centered on mapping the static properties of objects [7,8,9,10,11,12]. Basically, the mapping process of static properties of each class results in one or more relations with referential integrity among them. Typically, these static properties can be atomic, structured and multi-valued attributes (Oracle VARRAY), and references to other objects/classes (Oracle REF clause). Few authors address the mapping of some dynamic properties [13,14,15,16].

According [13], converting an object model to a set of declarations in a typical object-oriented language is relatively straightforward for most programmers. Basically, class declarations correspond directly to object models, and

associations can be implemented as stand-alone objects or as pointers from object to object. Considering that many OODBMS are normally based on some object-oriented programming languages, it is possible to infer that the designers have the same level of difficulty to perform the mapping process from UML classes to OODBMS description models.

[26] present a discussion of mapping ODL (Object Definition Language) and E/R Diagrams to Relational model. ODL is a proposed standard language for specifying the structure of databases in object-oriented terms. It is an extension of IDL, a component of CORBA. Basically, it is discussed the mapping of ODL atomic and non-atomic attributes, ODL type constructors (set, bag, array and list) and ODL single and multi-valued relationships. The mapping of E/R designs to relational model is similar to the discussion presented by [27]. Of course, both [26] and [27] discuss the mapping of static properties described by E/R diagrams.

[15] briefly presents a mapping process from UML class diagrams to object-relational databases, based on SQL:1999 [17]. In this sense, they present some algorithms to transform UML class diagrams removing semantically overloaded UML elements, and to map this latter UML class diagrams to object-relational description model SQL:1999. According [15], a semantically overloaded element plays multiple roles rather than just one, and may cause undesirable data redundancy into the resulting database implementation. The implementation model written in SQL:1999 is normalized according to the Nested Normal Form – NNF [16]. The NNF accounts for certain types of redundancy in O-RDB tables. Also, [15] discusses the problems of creating an arbitrary set of triggers related to the resulting relations of the process of mapping each UML class. The discussion is based on **termination, confluence** and **observable determinism** properties from active database systems [28]. It is presented an algorithm to evaluate if the resulting set of triggers, created by the system designers, satisfies the three properties above.

Specifically for mapping dynamic aspects described by state diagrams, [13] presents an implementation model for each class with dynamic properties, composed by a hierarchy of classes where each of them corresponds to one state of the class. Using this model to implement classes into relational databases imply on create a set of relations interrelated by foreign keys. Also, one state transition is done deleting the object from its current relation and recreating it into the new relation. For a real information system, where several classes have dynamic properties described by state diagrams, the complexity of the implementation may grow exponentially.

5. Conclusions

This paper presents a method for mapping dynamic properties of UML classes, described by UML state diagrams, to the relational database model using triggers. The mapping method presented permits the automatic building of the triggers for each database relation that has dynamic properties described by state diagrams. The method usefulness has demonstrated considering UML classes only with **atomic** attributes and UML state diagrams with **states** and **composite states** (without concurrent sub-states), **events**, **simple transitions** and **transitions to and from composite states**. But it is suitable for different class constructs as well. Our approach has been tested with a real case study with reasonable complexity and exhausted tests have been executed on the implementation.

The advantages of our approach are: (1) reducing the amount of programming efforts for an information system due to the implementation of a larger parcel of the dynamic properties, described by state diagrams, into the DBMS model, and (2) improving robustness of the databases against undesirable changes. We believe this method is capable to improve the quality of computational solution for information systems, implemented using relational DBMS. In terms of limitations of our method, we are considering only the dynamic properties intra-objects.

5.1 Future Work

Currently, we are working to extend the method towards supporting UML classes with type constructors (record, list, etc.) and references to other objects, and supporting other state diagrams elements, mainly, concurrent sub-states and history indicators. We have developed some prototypes to support concurrent sub-states and history indicators. Also, we are studying different types of CASE tools to identify an adequate environment to incorporate computer-supported functionality to automatically build the triggers. This study started by examining available UML tools, but we intend to analyze other environments.

References

[1] PRESSMAN, R. *Software Engineering*. New York: McGraw-Hill, 1987.

[2] DEMARCO, T. *Structured Analysis and System Specification*. Englewood Cliffs: Prentice Hall, 1989.

[3] YOURDON, E; CONSTANTINE, L. *Structured Design*. Englewood Cliffs: Prentice Hall, 1979.

[4] KOBRYN, C. UML 2001: A Standardization Odyssey. *Communications of ACM*. V. 42. Nº 10. Oct.1999.

[5] BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. *UML Users Guide***.** Addison-Wesley, 1999.

[6] BRODIE, M. L. *Migrating Legacy Systems: Gateways, Interfaces the Incremental Approach*. San Francisco, CA: Morgan Kaufman, 1995.

[7] AMBLER, S. W. *Mapping Objects to Relational Databases*. AmbySoft Inc. November 1998. 32p. Online. Available in Internet at http://www.Ambysoft.com/mappingObjects.pdf

[8] BROWN, K. *A Pattern Language for Object-RDBMS Integration*. "The Static Patterns". Knowledge Systems Corp. Cary-NC, USA, 1999. Online. Available in Internet at http://www.ksccary.com/Page15.htm

[9] DORSEY, P.; et al. *Oracle8 Design Using UML Object Modeling*. Mcgraw-Hill, Dec. 1998.

[10] HEUSER, C. *Projeto de Banco de Dados***.** (Database Design) Porto Alegre: Sagra-Luzzatto, 1999. 204 p. (in Portuguese)

[11] KELLER, W. *Mapping Objects to Tables, a Pattern Language*. Wien, Austria. Online. Available in Internet at http://www.sdm.de/g/arcus/

[12] RUMBAUGH, J. et al. *Object-Oriented Modeling and Design*. Englewood Cliffs, Prentice Hall, 1991.

[13] RUMBAUGH, J. Controlling code. How to Implement Dynamic Models. *Journal of Object-Oriented Programming*, SIGS Publications, May 1993. p.25-30.

[14] YACOUB, S. M. et al. *A Pattern Language of Statecharts*. West Virginia University. Computer Science and Electrical Engineering Department, and in Proceedings of PLOP-98 Conference.

[15] MOK, W. Y. *On Transformations from UML Models to Object-Relational Databases*. In: *HICSS'34- Hawaiian International Conference on System Sciences* (Maui, Jan/2001). Los Vaqueros – CA: IEEE Press, 2001.

[16] MOK, W. Y; Embley, D. W. A Normal form for precisely characterizing redundancy in nested relations**.** *ACM Transactions on Database systems*, V. 21, N° 1 p.77-106, March 1996.

[17] ANSI/ISO/IEC 9075 – 1999. *Database Language - SQL, Part 2.* Washington – DC: ANSI, 1999.

[18] OMG. *Unified Modeling Language, V.1.4: Part 9 – Statechart Diagrams***.** Needham – MA, USA: OMG, 2001. pp. 3-136 – 3-156.

[19] JACOBSON, I., BOOCH, G., RUMBAUGH, J. *The Unified Software Development Process*. Reading – MA, USA: Addison-Wesley, 1999. 463 p.

[20] HAREL, D. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming***.** Amsterdam, p231-274, June 1987.

[21] HAREL, D. On Visual Formalisms. *Communications of the ACM*, V. 31, nº 5, p. 514-530, May 1988.

[22] COHEN, D. *Introduction to Computer Theory*. 2.ed. New York, NY: J. Wiley, 1997. 838p.

[23] POHLMANN, O.; et al. Em Direção a Criação de uma Biblioteca Digital na PUCRS: Uma Experiência Prática. (Towards a creation of a digital library at PUCRS) *II Seminário Internacional de Bibliotecas Associadas a UNESCO.* Cienfuegos, Cuba. 1998. (in Portuguese).

[24] BARBIERO, E. et al. Workflow para Construção de Acervo Digital via WEB (Workflow to build a WWW digital library). *ICECE 2000* São Paulo: SENAC, 2000.p. 125-128. (in Portuguese).

[25] MACIEL, J. M. C. *Mapeamento de Propriedades Dinâmicas de Objetos, descritas por Diagramas de Estados UML, em Bancos de Dados Relacionais*. (Mapping Dynamic Properties of Objects, Described by UML State Diagrams, on Relational Databases). Porto Alegre: PPGCC-PUCRS, 2001. (MSc Dissertation, in Portuguese)

[26] ULLMAN, J. D.; WIDOM, J. *A First Course in Database Systems*. New Jersey: Prentice-Hall, 1997.

[27] BATINI, C. et al. *Conceptual database design*. Redwood City, CA: Benjamin/Cummings, 1992.

[28] AIKEN, A.; HELLERSTEIN, J. M.; WIDOM, J. Static analysis techniques for predicting the behavior of active database rules. *ACM Transactions on Database systems*, V. 20, N° 1, p.3-41, March 1995.