中華民國八十四年全國計算機會議論文集
Proceedings of National Computer Symposium 1995

將物件導向資料庫轉為考慮行為之全時程資料庫
MAKING AN OODB A BEHAVIORAL TEMPORAL OODB

陳國棟 李勇賢 劉寶鈞 洪炯宗 戴建誠
Gwo-Dong Chen, Yeong-Hsen Lee, Baw-Jhinne Liu, Jorng-Tzong Horng, Jian-Cheng Dai*

國立中央大學資訊工程研究所
Department of Computer Science and Information Engineering
National Central University
Chung-Li, Taiwan, R.O.C.

*資訊工業策進會
*Institute for Information Industry

摘要

目前之物件導向資料庫僅儲存最新之資料，然而在許多應用上需要資料過去的歷史，為了解決這個問題研究者提出全時程資料庫的觀念。在資料改變的過程一定牽涉到物件的行為，因此資料庫不僅要儲存資料的歷史而且要儲存行為的歷史以使得使用者能掌握資料庫的整個過程。目前之全時程資料庫並未考慮資料行為歷史的處理。本文在於提出一個方法擴增物件導向資料庫處理行為及資料歷史的能力，為了達成這個目的，我們提出了資料模式，詢問語言，儲存系統，及一個前端處理器來達成。我們並在一物件導向資料庫管理系統上製作一考慮行為之全時程資料庫。

關鍵字 : 物件導向資料, 全時程資料庫, 行為朔模

Abstract

Current object-oriented databases only manage the newest data of objects which are stored in the databases. That imply they can not support those applications for which the history data of objects must be taken into account. An application not only need to record the data history of objects but also need record the behavioral history of objects which make the state of data changed. However, current temporal database only consider the data history of objects but do not take the processing of behavior history of objects into account. The purpose of this paper is to extend the ability of current object-oriented database such that the database have the capability to process the data history and behavioral history of objects.

Keywords : Object-oriented database, temporal database, behavioral modeling

1. Introduction

Many Database applications requires history of data elements to make proper decision. For example, the credit department needs past account history of a customer to decide how much they can loan to him. Moreover, the department may also need to know how the customer manage his accounts. This requires the information of history of how accounts' values changed, that is, behavior history of the customer. Additionally, when an database application program goes wrong, the designer need to locate the errors of the program. It would be much easier for the designer to debug the program if the database provides information of data history and behavior history of the application system.

To support providing data history, many researches [1,7, 9,10, 11] tries to add time and data history facilities to database management systems. This kind of systems are called Temporal DataBase system (TBS). Most of the works are done on Relation DataBase (RDB) system, because RDB is the most popular database management system at current time. However, Object-Oriented DataBase (OODB) systems are gaining more and more attention in recent years. The interface description of an object is captured by its data and behavior. Although the behavioral part is essential for an object, most OODB researchers and systems did not put emphasis on the behavior aspect of objects. Some of the reasons are conventional database systems are only focused on data management.

Nevertheless, when we consider providing facilities for dealing with data history, we should consider not only data history but also behavior history. The reason is that data changes are caused by object behavior. By considering object behavior, a database system can not only provide data history but also the causes of data changes. Since an object is captured by its data and behavior, the OODB is easier to incorporate facilities for data history and behavior history.

Since there are many commercial · OODB systems available and they providing facilities for object data management and object-oriented programming language interface, it is more feasible to try to extend these systems with facilities for managing data history and behavior history.

The goal of this paper is to make an OODB system a behavioral temporal OODB system.

- It must capture the data history, the event history, and the interaction between event and data. Thus we can get complete behavioral temporal information of an application.
- It must have the ability to retrieve data in accordance with the time which is given by users.
- It can retrieve objects according to the data history.
- It can retrieve the data history path of objects.
- It can retrieve objects according to the behavior pattern.
- It can retrieve the behavior pattern of objects.

To make an OODB into a BTOODB, we devised approaches for data model, query language, storage system, and preprocessor respectively which is described in the following.

- **Data model**

  We propose a data model In TORI model, which uses three views to describe the database, that is Object-Relationship view, Object-Behavior view, and Object-Interaction view.

- **Query language**

  We propose a behavioral temporal query language to process the temporal query and is called BTQL (Behavioral temporal Query Language) [2.] The BTQL is a pattern-based query language which is able to issue queries that involves data histories and event traces. BTQL can also process the query that contains the uncertain transition path.

- **Storage system**

  Our system is based on the C++ interface of GemStone OODBMS.Between users and GemStone OODBMS, we provide a behavioral temporal data management level to manage the behavioral and temporal data for users, Fig. 1 depicts this architecture. In this level we support data structure and functions for behavioral temporal OO database.

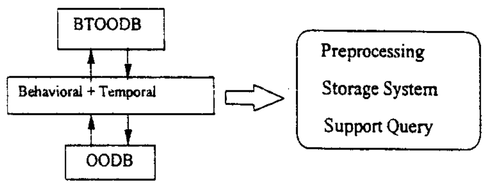


Fig. 1 The behavioral and temporal data management level.

- **Preprocessing for recording temporal data**

  A preprocessor is developed. Therefore, when an application program calls a method of object, it will call it via the database event handler. Thus, the database can capture the time of data history and event traces. We use the *event handler* to record required behavioral temporal data.

2 TORI Model

Since an object is defined by its data and behavior. We should capture both static (data) and dynamic (behavioral) aspect of objects. The behavior of an object can be decomposed into two parts: (1) object internal behavior and (2) object external behavior. Therefore, in TORI model, it includes three logical views of data. That is, object-relationship view, object-behavior view, and object-interaction view. With the object-relationship view, it mainly captures the static structure of data, but not monitor the dynamic evolution of data. The object-behavior view is incorporate with object-relationship view in order to capture the data evolution. Under the object-behavior view, all the dynamic information could be captured. In order to capture event traces or querying transactions, the object-interaction view must be included into the schema. TORI model integrates all the three views to capture the static data structure, data evolution, and event traces.The detail of TORI model can be found in [12].

3 Temporal OO Storage System

A behavioral temporal OO storage system must record all temporal and behavioral data of system such that the system can support users to retrieve behavioral and temporal data of objects. That imply the system should satisfy the following requirements.

1. Recording the data history.
2. Construct the data history path.
3. Recording the behavior history and construct the behavior history tree.
4. Connect the data history and behavior history to build the behavior and data interaction.

The contents of BTSS include the store of data history, data history path, behavior history, behavior history tree, the interactions between data and behavior, and support for behavior query. Section 3.4 shows the support for behavior query.

3.1 Data History and Data History Path

Data histories are the basic elements of a temporal database. For the temporal database, any version of an attribute should be recorded for temporal query and some other purpose. We use the approach called the *attribute versioning* approach [8, 3 ,5] to record temporal data of objects .Whenever the data of a temporal attribute changed, a new version of this attribute will be produced to record the data and the old data version is unchanged.

Some factors should be taken into account. The most important thing is how to make the data can be retrieved quickly. Another consideration is the data redundancy. Inevitably, the history data will cost large storage space. Once users are not interested in an attribute, they could set the attribute as non-temporal and just catch the current value of this attribute for saving the storage space. Fig. 2 shows the contrast between non-temporal object and temporal object.

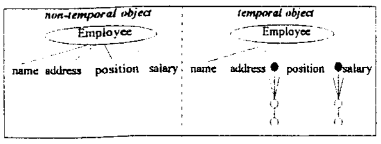
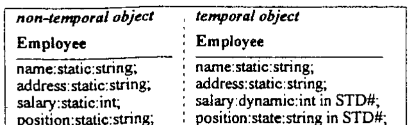


Fig. 2 The different structure between non-temporal and temporal Object

The storage structures of a non-temporal object and a temporal object are show in Fig. 3. Since the states of *salary* are varying and infinite, it must define as a dynamic temporal attribute. For the *position* attribute, the states of it is finite and can be expressed in some fixed stages, it must be defined as a state temporal attribute and own a STD (State Transition Diagram) to describe the transitions of its states.



non-temporal object	*temporal object*
Employee	**Employee**
name:static:string;	name:static:string;
address:static:string;	address:static:string;
salary:static:int;	salary:dynamic:int in STD#;
position:static:string;	position:state:string in STD#;

Fig. 3 The storage data structure of non-temporal object and temporal object

In our method, the storage structure of each temporal attribute is a link list of state entities. For storing history data of a temporal attribute, the structure is defined as below.

```
class state{
 char    *Title;     // The value of state.
 time    FromTime;
         // The time that an attribute transfer to
         //this state
 time    TillTime;
         // The time that attribute leave this state
 event   ChangeBy;
         // The event  that will make the state change
};
```

The data history of a temporal relationship is constructed in the same way as objects. While a temporal relationship is built, a link list of data history will be constructed at the same time.

Fig. 4 shows the data history path of employee. Where the temporal attributes *salary* and *position* of employee are associated with a data set which record all the versions of the attributes, and each attribute own a link list of history data.

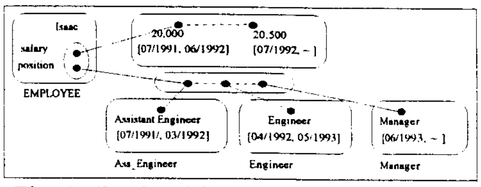


Fig. 4 The data history path of Employee

Fig. 5 show the class hierarchy of states. The terminal nodes are concrete nodes, which means the data are really stored in, and for the non-terminal nodes, there only are Oids which are a pointer to an instance of the concrete nodes.

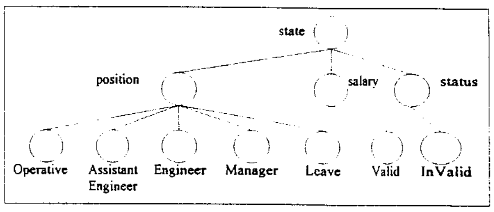


Fig. 5 The state class hierarchy

For temporal relationship, we conceive it must contains at least three basic elements. Evidently, the objects (at least two) that participate in the relationship is necessary. The third is the status of this relationship which indicates the relationship is valid or not and is varying over time.

Fig. 6 is the data history path of relationship. For which while the participates of an relationship are changed, a new instance of relationship will create and set the status of previous relationship become Invalid.

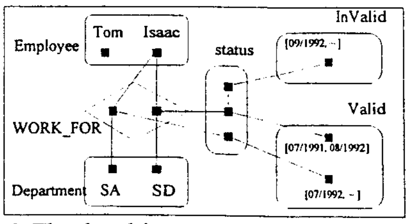


Fig. 6 The data history path of WORK_FOR

3.2 Behavior History and Behavior History Tree

Data history just record the state transition data, while to complete record the behavior of objects in a system, the circumstances of method invocation must also be recorded.

The data of behavior history contains function call and member function invocation. While an object want to get some information from other objects or just invoke a member function of another object, this object must request the services of other objects by function calls. The member function invocation is the function call insides an object. In our system, any function call must be recorded in the behavior history since without the behavior histories, we can not know the reason why makes data chage and can not support queries about the behavior of objects.

To capture the behavior history of an object, how to record the method invocation is difficult. Since a method of an object can only be invoked by the object itself. If an object want to invoke a member function of other objects, to request the member function supported by those objects is the only method. Therefore, it is impossible to know who invoke the member function of an object when another object invoke the function.

An *event handler* is proposed to manage the method invocation and construct these behavior histories. When an event wants to invoke other events (function call), this is to be done via the *event handler*, and the *event handler* will store these information into the database. Those data recorded by the *event handler* is the data of behavior histories.

For an instance of the behavior history, it must keep the following information. A title record the name of the function who had been assigned to execute. The event time record the time of function been invoked. The parameters and return values are also record, where parameters are transfer from the calling function and return values are the results that will return to the calling function. The state which is induced by this event is also recorded which captured the interaction between data and event. Additionally, an event are exist to signify the event who invoke this event and will construct event traces as a reverse link list. The reason we use this strategy is for the consideration of implementation, since the number of events who was invoked by a event is varying and is difficult to record. The structure of class event is define as

```
class event{
  char      *Title;
     // The name of this event(function);
  time      EventTime;
     // The time for which the event was
     //happened
  params    Parameters;
     //The parameters received from
     //induced function
  params    ReturnValues;
     // The result that will return to parent
     //functions
  event     InvokeBy;
     // The event who invoke this event.
  state     InduceState;
     // The state induced by this event.
};
```

Construct the Behavior History Tree (Event Trace)

In our system, instances of behavior history are stored in a single set which connect the correlative instances to construct the behavior history tree. Fig. 12 illustrates the behavior history tree (event trace) of *employee*. Where while an employee enters a company, the event "EnterCompany" induces three functions, that is "Promote", "ChangeSalary", and "SetDepartment" to initialize the data of an employee. All these function calls will constitute the behavior history tree.

The member function "ChangeSalary" of employee is a trigger function, when the condition of a trigger becomes true, the actions associated with it are executed. As introduced in O++ [6], a trigger must add to those member functions who have the possibility to fire this trigger with the method of event call. Since the member function "EnterCompany" and "Promote" can fire the trigger "ChangeSalary" as displayed in Fig. 7. That means, while method "Promote" and "EnterCompany" been executed, they also need to execute the trigger to check whether the salary will be change or not.

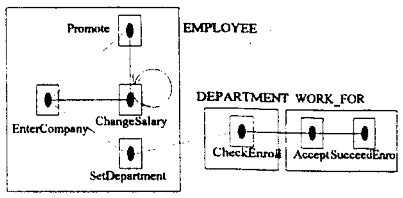


Fig. 7 Behavior history tree

3.3 Behavior and Data Interaction

For a temporal system, the state transition construct the data history as describe in section 3.1, the event trace build the behavior history as explain in section 3.2. It is obviously that a state transition is always induced by an event (function or trigger), and the relationships between data history and behavior history are defined as the behavior and data interaction of a temporal system.

The data history path and behavior history tree can not store independently. There should exist links to connect the event traces and state transitions therefore the behavior and data interaction can build.

Fig. 8 depicts the behavior and data interaction of an employee. In this figure, the state transitions are incorporated with event traces.

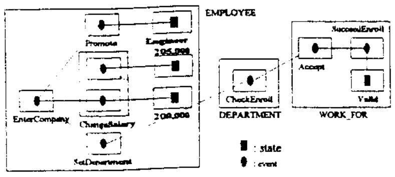


Fig. 8 Behavior and data interaction

3.4 Support for Behavioral and Temporal Query

For those approaches that do not record the event traces, to accomplish the queries that contains the behavior patterns is impossible. In our approach, the event traces are incorporated into temporal database, that means the behavioral queries are practicable in our system. Traditional non-temporal queries are also support in our system. This section demonstrates the queries that our system have support, where the S, D, and E presented in BTQL are represent the type of attributes is state, dynamic, and event relatively.

3.4.1 Behavioral Query

Display the employees whose position had promote to manager and then leave the company. This query demonstrates for a given behavior pattern we can get the objects which satisfy the behavior pattern.

```
SELECT employee.name
FROM   EMPLOYEE employee
WHERE employee.position WITH *
-E(promote)-S(Manager)-E(Leave)-*
```

3.4.2 Temporal Query

Retrieve the data history path of employee Christia's salary during the time interval from 01/1994 to 12/1994. This query is a demonstration to show the data history path of an object and with the filter to select attribute which we want to retrieve the data history path.

```
SELECT S.Title
FROM   EMPLOYEE employee
WHERE employee.name =
    Christia"
    AND employee.salaryWITH*D-*
    AND  S.FromTime OVERLAP
    [01/1994, 12/1994]
    AND  S.TillTime OVERLAP
        [01/1994, 12/1994]
```

4 System Implementation

The system implementation will introduce how to convert a database schema based on TORI model into a behavioral temporal object-oriented database. We use an ODL handler and an ODL code generator to accomplish this. An database of employee is used to demonstrate the processing of implementation.

4.1 System Architecture

In this system, the process to construct a behavioral temporal OO database was separate into three steps. The first is build the schema of database, where the attributes, state transition diagram, constraints, triggers, and the relationships should be well defined. After the schema definition finished, the ODL handler begins to progress and produces a temporary program that use the literal statement to describe the behavioral temporal object-oriented database schema.

The purpose of ODL handler is to convert the system from conceptual level to implementation level. In the conceptual level, what user need to do is describe the properties of classes and the relationships between classes. Implementation level, which provide the concepts that may be understood by end users but that are not too far removed from the way data is organized within the computer. Implementation data models hide some details of data storage but can be implemented on a computer system in a direct way[4].

We support the flexibility for user to define the special functions in this step, where user can add the pseudo function code to the temporary program which are not present in the schema, or modify the pseudo code. If user had defined schema well during the first step, then the modify can pretermit and execute the code generator to generate the C++ program.

After the program succeeded generate, use the GemStone C++ compiler to compile the program into executable behavioral temporal object-oriented database system. Fig. 9 depicts the architecture of this system.

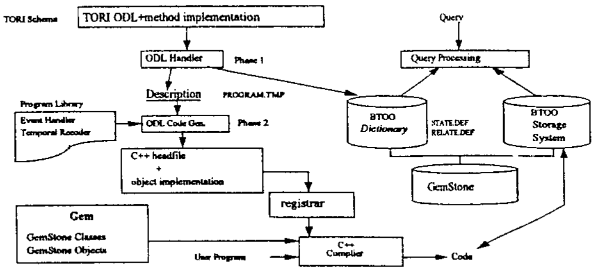


Fig. 9 System architecture

We use two phases to accomplish the system. The purpose of this architecture is for the flexibility of users, the reliability for system, and the simplicity for processing. After phase 1, the pseudo program code generated and users can check or modify it to satisfy the requirement of themselves. For the reason of reliability, the two phases architecture can analyze more accurately than one phase architecture. Since the one phase architecture generate the C++ program code from schema diagram directly, it's evidently that the process of one phase architecture will more complex than two phases architecture.

4.3 ODL handler

The ODL handler is a preprocessor of the system, it extract objects, state transitions, member functions, and other information from the schema. In this step, three temporary files will be generated, that is "STATE.DEF", "RELATE.DEF", and "PROGRAM.TMP". All the information describe in the schema will extract to these temporary files. The metafile "STATE.DEF" store the information of interactions between state and event of the system. This file will be helpful while system want to calculate the next state of current state. "RELATE.DEF" store the information about the relation of class, set, and relationships in the system. This file is useful to make the temporal query process fast and easy. Besides, there exist another temporary file called "PROGRÁM.TMP". It is not an executable file, but will be helpful while the generating of executable program is progress. If need, user can modify or add instructions, or functions to this file.

The Algorithm of ODL handler

Table 1 describes the algorithm of ODL handler. This ODL handler will extract the components from schema. During the schema diagram construct, a file will create to store the log of traces while construct the schema diagram. The trace file contains a lot of descriptions that register the steps while construct the schema diagram. The algorithm of ODL handler will search the trace file and get information to build the temporary files that had mentioned above. The algorithm of ODL handler is list as follows.

```
while there still have other classes not proceed{
    select a class component that still not processed.
    get class name and attributes.
    add these data to "PROGRAM.TMP"
    while there have any STD not processed{        // Construct STATE.DEF
        select a STD that still not processed.
        extract the state and event information from STD.
        add the state and event information into "STATE.DEF".
    }
    add the data of member functions to "PROGRAM.TMP"
    while there still have other constraints or triggers{
        get the information of constraints and triggers.
        add the condition and action to the member functions that will be influenced
    }
}
while there have a relationship not proceed{        // Construct RELATE.DEF
    Get the relationship information and add into the "RELATE.DEF".
}
```

Table 1 Algorithm 1--ODL handler

The Metafile "STATE.DEF"

The metafile is a auxiliary file for executing the state transition. While the state attributes are going to promote, degrade, or transfer to next state, system can search this file and get the position of next state.

In the "STATE.DEF" file, the data are indicate in a triple form, and is leading by a name of class which owns these state transitions.

The Metafile "RELATE.DEF

There have two parts in this file. It is "CLASS" and "RELATIONSHIP", and the declaration of "CLASS" part can be expressed with the tuple<*ClassName* ,*SetName*>.For example, the <"EMPLOYEE", "EMPLOYEESET" > means the instances of class EMPLOYEE are stored in the set named EMPLOYEESET. Of course, users can declare the name of set as "PERSONSET" or other names as they like. We must notice that the name of set can not as the same of class name, since the OODBMS will be confused and induce an error message as in GemStone OODBMS.

In like manner, the "RELATIONSHIP" part can be expressed with the triple <*Relationship, Class 1, Class 2*>. The first component is the name of relationship. The last two components are the class who had participate in the relationship. For example, the tuple.

To generating the relational table for classes is intuitional from the schema diagram. With the ODL handler algorithm, while a class been selected the name of the class will known at the same time and the name of set which store the instances of this class has verified.

The Pseudo Program Code "PROGRAM.TMP"

A class in "PROGRAM.TMP" contains six parts static, dynamic, state, constraint, trigger, and member. The static declare the static attributes and is list in table 2.

```
static:                // Define static attributes.
    char*      sno;
    char*      name;
    PERSON spouse;
    ........
```

Table 2 The declaration of Static attributes

Table 3 shows the declaration of dynamic attributes.

```
dynamic:             // Define dynamic attributes.
    int      salary;
```

Table 3 The declaration of Dynamic attributes

The state part declare the state attributes and is list in table 4.

```
state:              // Define STATE attributes.
    STATE   position;
```

Table 4 The declaration of State attributes

Constraint and trigger describe the conditions and actions of constraints and triggers. A constraint must follow the attribute for which the constraint is working on and is list as table 5.

```
constraint:                     // Define constraints.
    (spouse):                   // The name of attribute
        (spouse == NULL) || (this == spouse->spouse) :
        [spouse->Spouse(this);]
trigger:                        // Define triggers.
    month >= 7 && SFlag:
        [ChangeSalary();
        SFlag = !SFlag;]
    month >= 8 && !SFlag:
        [SFlag = !SFlag;]
```

Table 5 The declaration of Constraint and Trigger

The latest part is the declaration of members which describe the member functions of a class.

```
class EMPLOYEE{
    ........
    member:                              // Define member
functions.
        void Spouse(person):
            [spouse=person;]
        void Promote():
            [position=Get_Next_Position("PERSON", position,
"Promote");]
};
```

Table 6 PROGRAM.TMP

4.4 The ODL code generator

Before to processing the ODL code generator, we should define some basic classes for the system. For a temporal database, we abstract four basic classes, that are "METADATA", "METASTATE", "STATE", "EVENT", and "RELATION". Where the "METADATA" and "METASTATE" is the definition of metadata. The purpose for us to define these classes are to extend the OODBMS into behavioral temporal OODBMS.

4.4.1 Definition of Basic Classes

The first basic class is the state class, which is used to record the data of temporal attributes. The event which makes the position transfer to next state is also been record. A state attribute must carry the data as follows.

1. **The value of this state, it may be a string or other types.**
2. **The event which makes the status transfer to next state.**
3. **The time duration for which the attribute stays in this state, and contains a begin time and an end time.**

As the manner of state class, the event class also use the recursive declaration to construct the event traces. Other important note is the link method of events. Since it's hard to connect from an object to many objects, we use the back trace method to resolve this problem. That means an event will point to only one event that make it execute, and an event can be pointed by many events where those events were induced by this event. An event attribute must carry the following data.

1. **The title of this event.**
2. **The happen time of this event.**
3. **The name of function which induce this event.**
4. **The parameters which are transferred from the function that induce this event.**
5. **The return values which will return to the applied function.**
6. **The state which is induced by this event.**

To reduce the complexity of the system to process the relationship, all the multi-degree relationships in our system are decomposed to one-degree relationship. That means a relationship is always constructed by two objects and has at least one attribute to record the status of this relationship (Valid or Invalid). Any other relationships that carry more temporal attributes can derive from this class by inherit this basic class. A relationship must carry the following data.

1. **Two objects who are the participators of this relationship.**
2. **The status of this relationship and is one of valid or invalid.**

4.4.2 The Algorithm of ODL code generator

The following algorithm will going to extract the class declarations and method declarations from the "PROGRAM.TMP". Since in the temporary file, all the messages had well defined, to extract the classes and methods will become easy than the ODL handler. Table 10 is the algorithm to accomplish this, where contains two major parts. The first is to extract the class declarations, that is the type define of attributes, and the signature of member functions. Second, the methods' declaration will construct. As describe in 4.3.4, the instructions of constraints only insert to the member functions which have the possibility to change the value of the attribute that set in the constraint. Comparing with the constraints, triggers are manipulated in a different manner. To imitate the function of trigger, the instructions of triggers must add to each member functions of this class. While a member function executes, the triggers can execute as well.

```
construct the structure of codefile and headfile.
while there still have other classes{
    Select a class name{
        construct the structure of this class      // construct head file
        get static attributes and store it to the private part.
        get dynamic and state attributes and store to the public part.
        store the primitive constructors and destructor.
        add the declaration of member functions.

    // construct code file
        read the information of method declaration from "PROGRAM.TMP".
        write the information of methods to the code file.
        add constraints to relative member functions.
        add triggers to all member functions.
  }}
```

Table 7 Algorithm 2 -- ODL code generator

6 Conclusion

To make an existing OODB a behavioral temporal object-oriented database system, we devised a data model and query language first. Based on the proposed data model and query language, we develope and implement a storage system and a preprocessor for object interface and implementation programs which was built on GemStone object-oriented system. A query processor then implemented on the storage system. This making GemStone into a behavioral temporal object-oriented system.

The extra burden for a database designer caused by the extension is that the designer needs to describe the state machine diagram for temporal attributes. Thus, developing effort on the behavioral temporal database system is about the same as on an existing OODB. By building application on an system, users of the application are able to query behavioral and temporal aspect of data history. Besides, application developer can use the system as a debugger tool for locating problems in the application system.

Our system do require extra processing time and extra storage space. However, owing to the rapid decreasing of the storage media cost and improvement of the processing computation speed, the storage requirement and performance is still acceptable, since the behavioral temporal database system can offer information that can not provided by existing database systems.

References

[1] J. Clifford and A. Croker, "The historical relational data model (HRDM) and algebra based on lifespan.", *Proceedings of the third International Conference on Data Engineering*, pp. 528-537, Los Angeles, CA, February 1987.

[2] G.D. Chen, C.C. Liu, "A Behavioral Temporal Query Language"

[3] J. Clifford and A.U. Tansel, "On an Algebra for Historical Relational Databases: Two Views.", *Proceedings, ACM SIGMOD Conference*, 1985.

[4] Ramez Elmasri, Shamkant B. Navathe, "Fundamentals of Database System", *The Benjamin/Cummings Publishing Company, Inc.*, 1989.

[5] R. Elmasri, G. Wuu, "A Temporal model and Language for ER Databases.", *Proceedings, Sixth IEEE Data Engineering Conference*, February 1990.

[6] N. Gehani, H.V. Jagadish, "Ode as an Active Database: Constraints and Triggers", *Proceedings of the 17th International Conference on Very Large Data Bases*, pp. 327-336, Barcelona, Sep. 1991.

[7] S.K. Gadia, S.S. Nair, "Temporal Databases: A Prelude to Parametric Data.", Temporal Databases: theory, design, and implementation, pp. 28-66, 1993.

[8] S. Gadia, C. Yeung, "A Generalized Model for a Temporal Relational Database.", *Proceedings, ACM SIGMOD Conference, 1988.*

[9] N.A. Lorentzos, "The Interval-extended Relational Model and Its Application to Valid-time Databases.", Temporal Databases: theory, design, and implementation, pp. 67-91, 1993.

[10] A.U. Tansel, "Adding time dimension to relational model and extending relational algebra.", *Information System*, 11(4):343-355, 1986.

[11] Gene T.J. Wuu, U. Dayal, "A Uniform Model for Temporal Object-Oriented Database",*IEEE, 1992.*

[12] Chen-Jun, Liu "A behavioral temporal data model and it's query language" Master Thesis 1995 Dept. of CSIE National Central University.