Operators of the Temporal Object System and their Implementation

A. Shah[1], F. Fotouhi[2], W. Grosky[2], S. H. Shah Khan[1], and J. Al-Muhtadi[1]

[1] Department of Computer Science
King Saud University
P.O. Box 51178. Riyadh, Saudi Arabia

[2] Department of Computer Science
Wayne State University
Detroit, Michigan, 48202, USA

Abstract

We proposed a Temporal Object System (TOS) which maintains changes to both the structure and the state of an object in a temporal fashion (Shah, 1992; Fotouhi et al, 1992a; Fotouhi, et al, 1992b; Fotouhi, et al, 1994). Objects in TOS are referred to as *temporal objects* and are allowed to evolve over time. A collection of temporal objects which share the same set of common properties is grouped into a family. A temporal object that can be defined by using the local knowledge of a family is referred to as an *offstage object* (Shah et al, 1993a). We also discussed the renovations of both temporal complex objects and offstage objects in (Fotouhi et al, 1992; Shah et al, 1993a; Fotouhi et al, 1994).

This paper is a continuation of the work reported in (Fotouhi et al, 1994), and now we report on the operators of the temporal object system (TOS) and their implementation. These operators are grouped into three different modules of the TOS based on their relevant functions. These modules are: *Object Manager* (or Object Module), *Family Module*, and *Root of TOS (RTOS) Module*. The important module is the Object Manager (OM) that consists of basic operators. The modules provide a facility for defining a simple temporal object and later to add a stage in the temporal object. The other operators are grouped into the two other modules and are

referred to as RTOS module and Family module. We have implemented these operators using the SELF version 4.0 programming language on a SUN Sparc Workstation running Solaries 2.4.

Key Words: object-oriented databases, temporal database, temporal objects, object manager

1. Introduction

In most of the existing relational database systems, data objects are stored in a non-temporal fashion. That is, when the value of an attribute changes, the old value is replaced by the new value. Thus, only the latest state of an object resides in the database. However, for many database applications such as Computer-Aided Design/Computer-Aided Manufacturing (CAD/CAM), Computer Aided Construction (CAC), etc., it is not appropriate to discard old information. In these cases it is necessary to associate time values with data to indicate the time for which the data is valid. A time dimension is added to a database either at the attribute level (Clifford, 1982) or the tuple level (Gadia, 1991) to keep the history of a data object. Such a database is referred to as a *temporal database* (Dutta, 1989; Gadia et al, 1991).

Time in a temporal relational system is modeled as either a time point or a time interval. Both time models are considered equivalent (Dutta, 89). The value of time associated with a data object is determined by the system or assigned by the user. If a time value is assigned by the system, then it is referred to as a *physical time* such as transaction time, while if it is assigned by the user, then it is referred to as a *logical time* such as user-defined time (Ling et al, 1990).

Relational database and their temporal extensions are suitable for simple record-based applications, but are not suitable for engineering and other complex database applications due to their limitations in defining a complex object directly and at one place (Mair, 1986). Several types of object-oriented databases have been proposed to support the development of these application systems. In the object-oriented paradigm, an object is defined by two parameters: *structure* and *state*. The *structure* (SR) of an object provides the structural and behavioral capabilities to that object, and it is defined by a set of instance variables and methods. The *state* (ST) of an object assigns data values to the instance variables of the objects, and the methods

operate on them. A set of objects that share the same structure is referred to as a *class*. An object-oriented database is a collection of classes which are organized as a directed acyclic graph (DAG) or a simple directed graph (Kim et al, 1987, Kim, 1990).

In the existing object-oriented database systems, changes to the state of an object are maintained via *version management* (Agrawal et al, 1991; Katz et al, 1986). Also, structural changes are supported in most of the object-oriented database systems. Such changes to a class are referred to as *schema evolution* in the literature (Nguyen et al, 1989; Roddick, 1992). There are three possible scenarios of a class to change its structure. These are:

(type I) Adding new instance variables and/or methods

(type II) Deleting instance variables and/or methods

(type III) Updating an instance variable and/or a method

For type I changes, there is no loss of *knowledge* in a class because the previous knowledge of the class structure is also retained along with the new one. We define knowledge of an object of a class by the structure of the class. In the TOS environment, we define knowledge of a temporal object or a stage by the structure, the state, or both. Both (temporal object and stage) are defined later in this paper. On the other hand, for type II and type III changes, the history of the changes to a class structure is not readily available, as it is overwritten or deleted in the latest version of the class structure. Current object-oriented database systems keep only the current version of each class structure. *After* any one of the type II or type III changes, it is necessary to reload a previous version of the database to retrieve any information from a previous version of a class structure. It is a desirable feature of CAD/CAM, CAC, and other advanced engineering database applications to keep history of changes to the class structures (Shah et al, 1993). It helps in designing a new product by using previous products.

In (Fotouhi et al, 1992a; Fotouhi et al, 1994; Shah, 1992) we introduced a temporal object-oriented system (TOS) which maintains the history of changes to both the structure and the state of an object in a consolidated and elegant manner. We associated time (point model) to

both the structure and state of an object. Such an object is referred to as a *temporal object*. A temporal object evolves over time due to changes to its state, structure, or both. A set of temporal objects which share a common knowledge (i.e., structure and state) is referred to as a *family*. The TOS also facilitates the construction of a *complex family* which is an aggregation of temporal objects from various families. The objects in a complex family are referred to as *temporal complex objects* (for details see (Fotouhi et al, 1992b and Shah et al, 1993b)). A complex family increases the knowledge sharing (or reusability) of non-homogeneous temporal objects and their transportability from one family to another. A temporal object system (TOS) is therefore a collection of families which are defined at different time instances. The concept of *renovation* refreshes the knowledge of temporal complex objects and offstage objects by replacing their sub-objects and participant objects (they are defined later in Section 2), respectively, at any time instance.

In this paper, we extend our work that was reported in (Fotouhi et al, 1994), and define operators of the TOS. Function of each operator is different such as the creation of simple families, complex families, simple temporal objects, offstage objects, temporal complex objects, renovation of objects, and so on. The operators are grouped into the different modules which are referred to RTOS Module, Family Module, and Object Module (or Object Manager). We have implemented this module using SELF (version 4.0) programming Language. Currently these modules are doing basic functions such as the creation of simple families, simple temporal objects, stages, etc.

The remainder of this paper is organized as follows: In Section 2, we describe the TOS and its components. Section 3 gives syntax of the operators of the TOS and their description. In Section 4, we give the system architecture that implements the operators. We have implemented the operators in SELF programming language, Section 4 deals with the implementation details. Finally, in Section 5, we give our concluding remarks and future research directions.

2. Temporal Object System (TOS)

As mentioned earlier, the TOS is a collection of simple and complex families which are defined at different time instances. A family is a collection of temporal objects, and a temporal object is a collection of stages. Figure 1 shows a general schema of the TOS, where *RTOS* represents the root node of the system with n families, i.e., F_1, F_2 F_3,. . . F_n as its children. These families are constructed in the TOS at the time instances, t_1, t_2, . . ., t_n, respectively (see Figure 1). The figurative notations which are used in Figure 1 and in other figures are given in Appendix at the end of the paper.

In the next section we give a brief overview of the temporal objects which are the building blocks of the family, and then we discuss the concept of families.

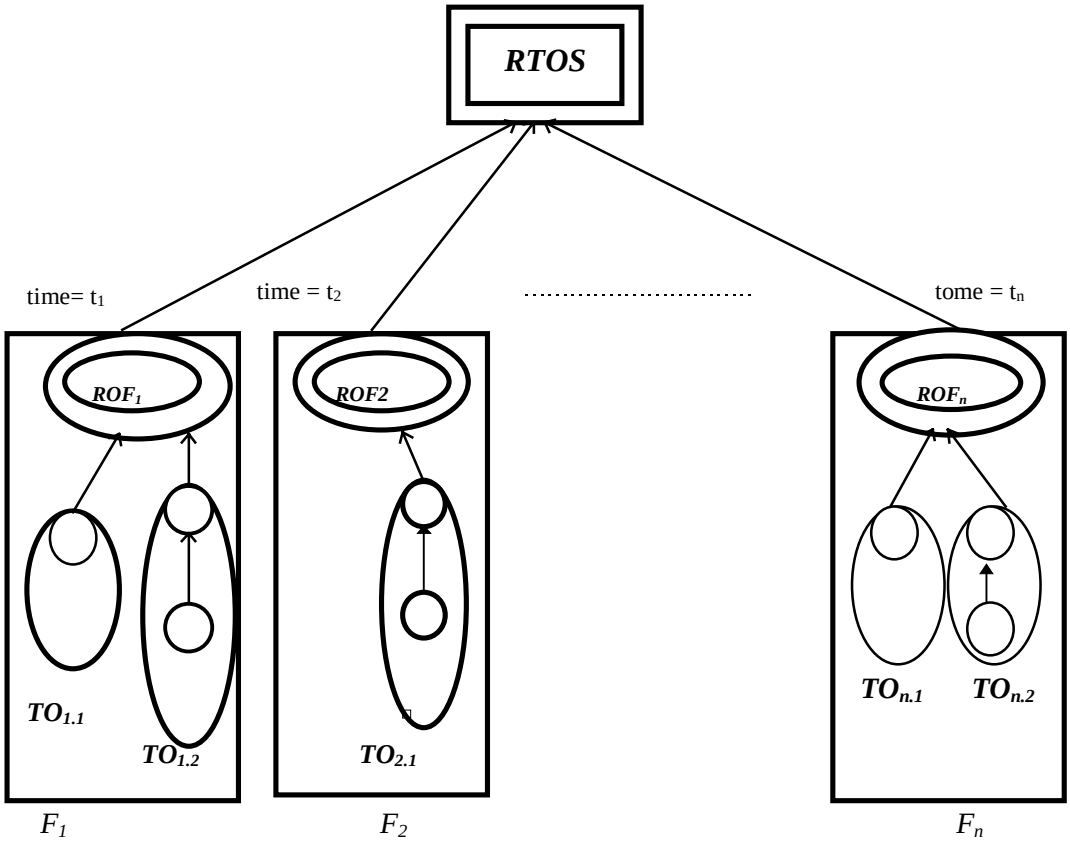


Figure 1: Schema of a Temporal Object System (TOS)

2.1 Time Dimension in TOS

A time dimension is associated with the creation of a stage, a temporal object, and a family in TOS. Time is explicitly defined by the user as an instance variable. While creating a

TOS application the data type of the time is defined. The data type of the time should be mentioned in the creation of all families, temporal objects, and stages in future. The granularity of time depends on the application domain. In the TOS, we use time point model (Dutta, 1989; Ling et al, 1990). A time point is referred to as a *time instance*. A time instance is a distinct and discrete point on the *time-line* and a dimension-less entity. Time interval is also used in the TOS to represent time duration between two time instances, for example, time-span of a stage and life-span of a temporal object. In this paper we use an abstract time *"year"* in each stage, temporal object and family for the sake of simplicity.

2.2 Temporal Objects

As mentioned in the previous section, an object is represented by its structure and state. With the passage of time an object may change its structure and/or its state. By associating time to both the structure and the state of an object, we can keep the history of changes to that object. Therefore, we define a temporal object (TO) to be an ordered set of objects which is constructed at different time instances. A temporal object is represented as $TO = \{(SR_{t1}, ST_{t1}), (SR_{t2}, ST_{t2}), \ldots, (SR_{tn}\ ST_{tn})\}$ where $t_i \leq t_{i+1}$ for all $1 \leq i < n$, and where the ordered pair (SR_{ti}, ST_{ti}) is the i-th object of the temporal object which is constructed at the time instance t_i, with structure SR_{ti} and state ST_{ti}. An i-th object of the temporal object is referred to as its i-th stage (Fotouhi et al, 1992a; Fotouhi et al, 1994; Shah, 1992). A new stage (or *current stage*) of a temporal object shares the structure and/or state from the previous stage, which is not defined in the new stage. Both current and previous stages are constructed as prototypes, defined in the prototype-based approach (Borning, 1986). In the prototype-based approach, a new prototype can be defined from an existing prototype by capturing the structure and/or state that is not defined in the previous prototype (Borning, 1986; Chambers et al, 1989).

A stage is maintained in a prototypical form, i.e., a structure, a state, or a combination of the two (Borning, 1986). For example, if a temporal object suffers a structural change, then the new stage of the temporal object captures only the structure change. A temporal object may also

be referred to as *an ordered set of stages*. For example, in Figure 2 the temporal object TO_a of the family F_i has n number of stages. The first and last stages of a temporal object are significant because they hold the initial and current knowledge of the temporal object. We refer to these stages as the *birth stage* (stage $S_{i.a}$ in Figure 2) and the *current stage* (stage $S_{n.a}$ in Figure 2) of the temporal object TO_a. The current stage (or the n-th stage) is the latest stage that is appended to the temporal object. A new stage is appended to a temporal object when a change occurs to the structure and/or state of the temporal object.

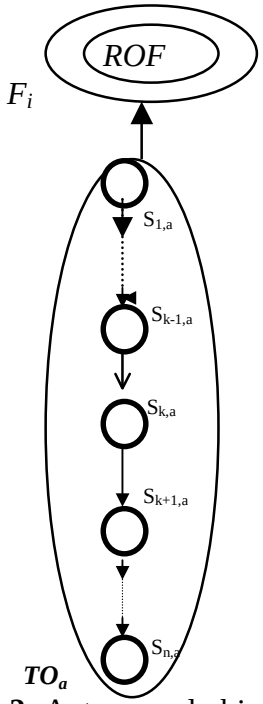


Figure 2: A temporal object TO_a

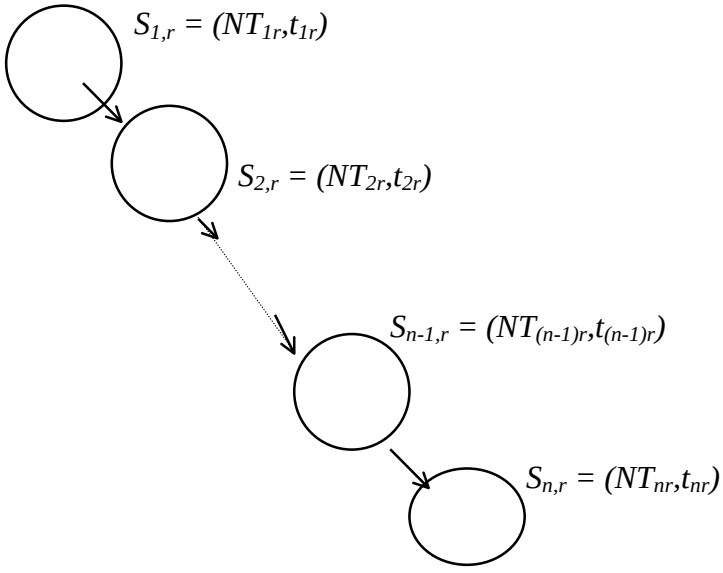


Figure 3: Temporal object TO_r with n number of stages

2.2.1. Temporal Parameters

An ordered sequence of stages of a temporal object is referred to as the *life-sequence* of the temporal object. The temporal object TO_r has *life-sequence* $\boldsymbol{L_r} = \{S_{1,r}, S_{2,r}, \ldots, S_{j,r}\}$ (see Figure 3) where $S_{1,r}$ is the birth stage and $S_{j,r}$ is the j-th stage (which is not the current stage) of the temporal object. The set $\boldsymbol{L}$ may also be defined as $\boldsymbol{L_r} = \{(NT_{1r},t_{1r}), (NT_{2r},t_{2r}), \ldots, (NT_{jr},t_{jr})\}$ where the ordered pair (NT_{ir},t_{ir}) shows the knowledge (non-temporal and temporal) of the i-th stage of the temporal object TO_r. The NT_{ir} is non-temporal knowledge of the i-th stage, which can be a structure, state, or both, and the time instance t_{ir} is the temporal knowledge of the stage. The set $\boldsymbol{\tau} = \{t_{1r}, t_{2r}, \ldots, t_{jr}\}$ is the temporal knowledge of all stages in the set $\boldsymbol{L_r}$. Note that the two subscripts used in temporal and non-temporal knowledge of the stages identify stage number in a temporal object and the temporal object, respectively. The set $\boldsymbol{\tau}$ of a temporal object satisfies the inequality $t_{ir} \leq t_{(i+1)r}$ for all $1 \leq i < j$. If the set $\boldsymbol{L_r}$ includes all the stages in a temporal object stage, then the set $\boldsymbol{L_r}$ is referred to as a *complete life-sequence* of the temporal object and denoted by $\boldsymbol{L}$. The complete life-sequence, $\boldsymbol{L}$, of the temporal object TO_r is given (see Figure 2) as follows:

$\boldsymbol{L} = \{(NT_{1r},t_{1r}), (NT_{2r},t_{2r}),\ldots,(NT_{ir},t_{ir}), (NT_{(i+1)r},t_{(i+1)r}),\ldots,(NT_{jr},t_{jr}), \ldots, (NT_{nr},t_{nr})\}$

The set $\boldsymbol{L_1} = \{(NT_{ir},t_{ir}), (NT_{(i+1)r},t_{(i+1)r}),\ldots,(NT_{jr}, t_{jr})\}$ for the temporal object TO_r is referred to as *partial life-sequence* of the temporal object if $\boldsymbol{L} \supset \boldsymbol{L_r}$, where $\boldsymbol{L}$ is the *complete life-sequence* of the temporal object. A complete life-sequence of a temporal object shows a complete life history of the changes that occurred to the temporal object.

Two terms, *life-span* for a temporal object and the temporal parameter *time-span* for a stage are defined to study their temporal behavior. The temporal parameters are also used later in temporal queries on temporal objects and their families. In Figure 3, the birth stage $S_{1,r}$ of the temporal object TO_r is created at the time instance t_{1r}, and later at the time instance t_{2r} the second stage $S_{2,r}$ is created. The time interval $[t_{1r}, t_{2r}]$ (where $[t_{1r},t_{2r}] = \{x \mid t_{1r} \leq x \leq t_{2r}\}$ is referred to

as a *closed interval* on both ends) between the two time instances t_{1r} and t_{2r} defines the *time-span* of the stage $S_{1,r}$. It is time gap (or time difference) between the creation of two consecutive stages in a temporal object. The interval $[t_{1r},t_{2r})$ is closed on the left end and open on the right end, i.e., $[t_{1r},t_{2r}) = \{x \mid t_{1r} \leq x < t_{2r})\}$. During the time-span $[t_{1r},t_{2r})$, the temporal object TO_r had only the birth stage $S_{1,r}$ in its life-sequence. All the temporal queries which lie in the time-span, will be targeted to the stage $S_{1,r}$, because the temporal object consists of only the birth stage during the time interval $[t_{1r},t_{2r})$. A pair of a time-span and an object identity reduces the search space of a temporal query. In Figure 3, the stage $S_{n,r}$ is the current stage of the temporal object, and the time interval $[t_{nr}, now]$ which is a closed interval on both sides of the time interval, is the *time-span* of the current stage. The time interval $[t_{1r},now]$ is referred to as *life-span* of the temporal object at time instance *now*. Note that the time instance *now* refers to *present* time.

A temporal query on the temporal object TO_r (see Figure 3) with a time instance t_{kr} is a temporally valid query if $t_{kr} \varepsilon [t_{nr}, now]$; otherwise the query is temporally invalid. In the case in which the query is temporally valid, it may start its search from the current stage $S_{n,r}$ based upon the time value of the time instance t_{kr}, and the knowledge of the previous stages S_{n-1},r , $Sn-_{2,r}$, ...,$S_{2,r}$, $S_{1,r}$ may also be used in answering the query. If $(t_{kr} > now)$ is true or $(t_{kr} < t_{1r})$ is true, then the temporal parameter t_{kr} of the query is temporally invalid. The life-span of a temporal object and time-spans of a stage are two closely related entities. The time-span of each stage of the temporal object TO_r is given (see Figure 3) in the following table:

Stage	Time-Span
$S_{1,r}$	$[t_{1r}, t_{2r})$
$S_{2,r}$	$[t_{2r}, t_{3r})$
$S_{i,r}$	$[t_{ir}, t_{(i+1)r})$
.	
$S_{n,r}$	$[t_{nr}, now]$

The *life-span* of the temporal object TO_r is given as follows:

life-span = {[t_{1r} , t_{2r}), [t_{2r}, t_{3r}),.....[t_{ir}, $t_{(i+1)r}$),.....,[t_{nr}, now]}, a set of adjacent time intervals.

The life-span of the temporal object can also be denoted as a single time interval *[t_{1r}, now]*.

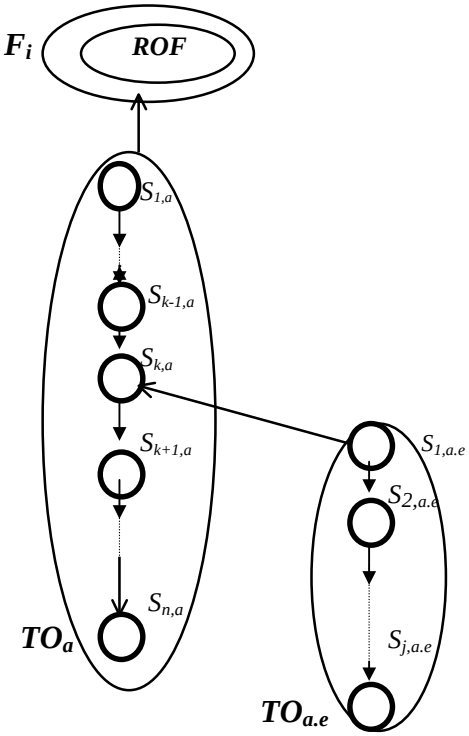


Figure 4: Offstage object $TO_{a.e}$ from temporal object TO_a

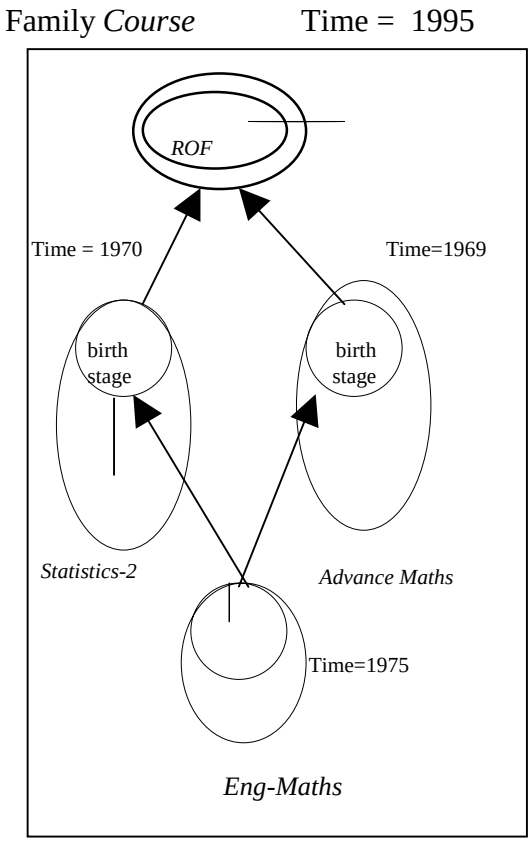


Figure 5: Offstage object *Eng-Maths* in the family *Course* - an example

Now we define the relationships among the temporal parameters; *time-span* and *life-span* of a family, temporal object, and stages. Assume that there are *n* number of temporal objects in a family F_f and TO_i is its the *i*-th temporal object. The temporal object TO_i has *j* number of stages. Time-span of the stages of the temporal object is the set $\{[t_{1i},t_{2i}), [t_{2i},t_{3i}), ..., [t_{ji}, now]\}$, and the life-span of the temporal object will be $[t_{1i},now]$. Similarly, we can enumerate these temporal parameters to the other temporal objects of the family. The life-spans of all n temporal objects of the family F_f at a time instance *now* can be enumerated as the set$\{[t_{11},now], [t_{12},now],.....[t_{1n},now]\}$. Each member of the set corresponds to the set of temporal objects $\{TO_1, TO_2,....TO_n\}$ of the family F_f, respectively. Consider the set $\boldsymbol{T}= \{t_{11}, t_{12},.....t_{1n}\}$, the set of time instances when *n* temporal objects took their births. If in a family we assume that time instance t_{inf} be the *infimum* (Greatest Lower Bound- GLB) of the set $\boldsymbol{T}$, i.e., $t_{inf} \leq t_{1m}, \forall\, 1 \leq m \leq n$ (or $minimum\{t_{11}, t_{12}, \ldots, t_{mn}\}$), then the time interval $[t_{inf}, now]$ is called the life-span of the family F_f. For the *j*-th temporal object TO_j of the family F_f, the life-span $[t_{1j},now]$ of the temporal object lies in the life-span $[t_{inf},now]$of the family where $t_{1j} \geq t_{inf}$. Now consider the *k*-th stage $S_{k,j}$ of the *j*-th temporal object, then time-span $[t_{kj},t_{(k+1)j})$ of the stage is also contained by the life-span $[t_{1j},now]$ of the temporal object, i.e., $(t_{kj} \geq t_{1j}) \wedge (t_{(k+1)j} \leq now)$. Therefore, in general, for a given family of the TOS, the following *containment relationship* that is denoted by "$\subseteq$" *will be true* for a temporal object and its stages, and the family of the temporal object;

time-span of stage $\subseteq$ *life-span of object* $\subseteq$ *life-span of family*

2.2.2. Offstage Objects

A new temporal object can also be created in a family by sharing the knowledge from an existing temporal object of the same family. The new temporal object is referred to as an *offstage object*. In Figure 4 the temporal object $TO_{a.e}$ is an offstage object that is defined by sharing knowledge of the temporal object TO_a in the family F_i. The offstage object $TO_{a.e}$ starts by sharing knowledge of the temporal object TO_a from its stage $S_{k,a}$ of the temporal object (see Figure 4). The stage $S_{k,a}$, is significant in the definition of the offstage, this stage is referred to as

an *offstage*, and the temporal object TO_a is referred to as a *participant object* (for more details of an offstage object and offstage see (Shah et al, 1993a)). The subscripts used in the offstage object $TO_{a.e}$ represent the participant object TO_a and the offstage object, respectively. Also, we are using "period" instead of "comma" to make the difference between the notations of a stage and an offstage object. The offstage $S_{k,a}$ is the stage from where the offstage object $TO_{a.e}$ starts sharing the knowledge of the participant object. The set of stages $\{S_{k,a}, ..S_{k-1,a}, ...,S_{1,a}\}$ of the participant object TO_a is shared in the offstage object. In other words, the offstage object $TO_{a.e}$ has taken its birth at time instance $t_{1,a.e}$ by sharing the set of stages $\{S_{k,a}, ..S_{k-1,a}, ...,S_{1,a}\}$ of the participant object TO_a. Note that both the temporal object and the offstage object have different object identities.

***ROF*(Course)**
Instance-Variables:
{ time=1965
course-code,
prereq,
outlines
}
Methods:
{update**}**

(a) *ROF* of family *Course*

state
{time:1970,
course:STA-201
course-name:*Statistics-2*
prereq:STA-101
outline:Discrete&
Continuous Probs.
Time Series Analysis**}**

(b) Participant temporal object *Statistics-2*

state
{time:1969,
course:MAT501
course-name:*Advance-Maths*
prereq:MAT201
outline:Diff&integral equns.
Laplace Trans,
Fourier Trans}

(c) Participant temporal object *Advance-Maths*

state
{ time:1975,
participants: *Statistics-2* &
Advance-Maths
course:ECE661
course-name:*Eng-Maths*
prereq:SAT-101 & MAT201
outline:Diff. &integral eq
Laplace Trans,
Fourier Trans}

(d) Offstage object *Eng-Maths*

Figure 6: Details of offstage object *Eng-Maths* and its participant temporal objects

Now we find life-span of the offstage object $TO_{a.e}$ and the participant object TO_a. The set of time-spans of stages and life-span of the participant object TO_a are given as follows:

$\boldsymbol{T}_a = \{[t_{1,a}, t_{2,a}), [t_{2,a}, t_{3,a}),...., [t_{k,a}, t_{k+1,a}),...., [t_{n,a}, now]\}$ where $t_{1,a}$ was the time instance when the first stage (birth stage) of the temporal object TO_a was created, and $t_{n,a}$ was the time instance when the current stage of the temporal object TO_a was created. Here, the set $\boldsymbol{T}_a$ is the set of time-spans of all stages of the participant object, the time interval $[t_{1,a}, t_{2,a})$ is the time-span of the birth stage, and the time interval $[t_{1,a}, now]$ is the life-span of the participant object.

The set of time-spans of the stages and the life-span of the offstage object $TO_{a.e}$ are given, respectively, as follows:

$\boldsymbol{T}_{a.e} = \{[t_{1,a.e}, t_{2,a.e}) , [t_{2,a.e}, t_{3,a.e}),...., [t_{j,a.e}, now]\}$ and life-span of $TO_{a.e} = [t_{1,a.e} , now]$.

Figures 5 and 6 exhibit an example of the offstage object *Eng-Maths* which is constructed in the family *Course*. The offstage objects *Eng-Maths* is defined at the time instance *1975* by sharing knowledge of two participant objects *Statistics-2* and *Advance-Math* through the offstage $S_{1,statistics\text{-}2}$ and $S_{1,Advance\text{-}Maths}$, respectively (see Figure 5). Each participant object has one stage (birth stage) at the time instance 1975 when the offstage object was defined. Figure 6(a) shows the *ROF* of the family *Course*. The details of the stages $S_{1,statistics\text{-}2}$ and $S_{1,Advance\text{-}Maths}$ of the participant objects are shown in Figures 6(b) and 6(c), respectively. Figure 6(d) shows the birth stage of the offstage object *Eng-Maths*. A temporal condition $(Eng\text{-}Maths.1975 \geq Statistics\text{-}2.1970) \wedge (Eng\text{-}Maths.1975 \geq Advance\text{-}Maths.1969)$ must be true *before* the creation of the offstage object, where *1970* and *1969* are the time instances when the offstages of the participant objects were defined.

If an offstage object is sharing knowledge of only one participant object, then this is analogous to simple inheritance in the class-based approach, and if an offstage object is sharing knowledge from more than one participant temporal objects, then this is similar to the concept of multiple inheritance in the class-based approach. An offstage object has some similarities and differences with an object which is defined by using the class-based approach. For example, an offstage object in a family and an object of a class both share the same common knowledge *ROF* and class structure respectively. In a family, state of a temporal object can also be shared by an

offstage object, whereas, in the class-based systems it is usually not allowed. The concept of an offstage enhances the reusability of knowledge within a family, and this benefit is not available in the class-based approach.

2.3 Families of the TOS

The concept of a family is used to assemble a group of temporal objects sharing a common context. All temporal objects within a family can be handled in a similar fashion by responding uniformly to a set of messages. A set of structures and\or states (available at the time of defining a family) defines a *common context* of the family. The common context of a family is referred to as the *root-of-family* (*ROF*) where common knowledge about all its temporal objects is maintained (see (Fotouhi et al, 1992a; Fotouhi et al, 1992b; Fotouhi et al, 1994; Shah, 1992) for more details). Temporal objects of a family can be defined only *after* the construction of *ROF* of the family.

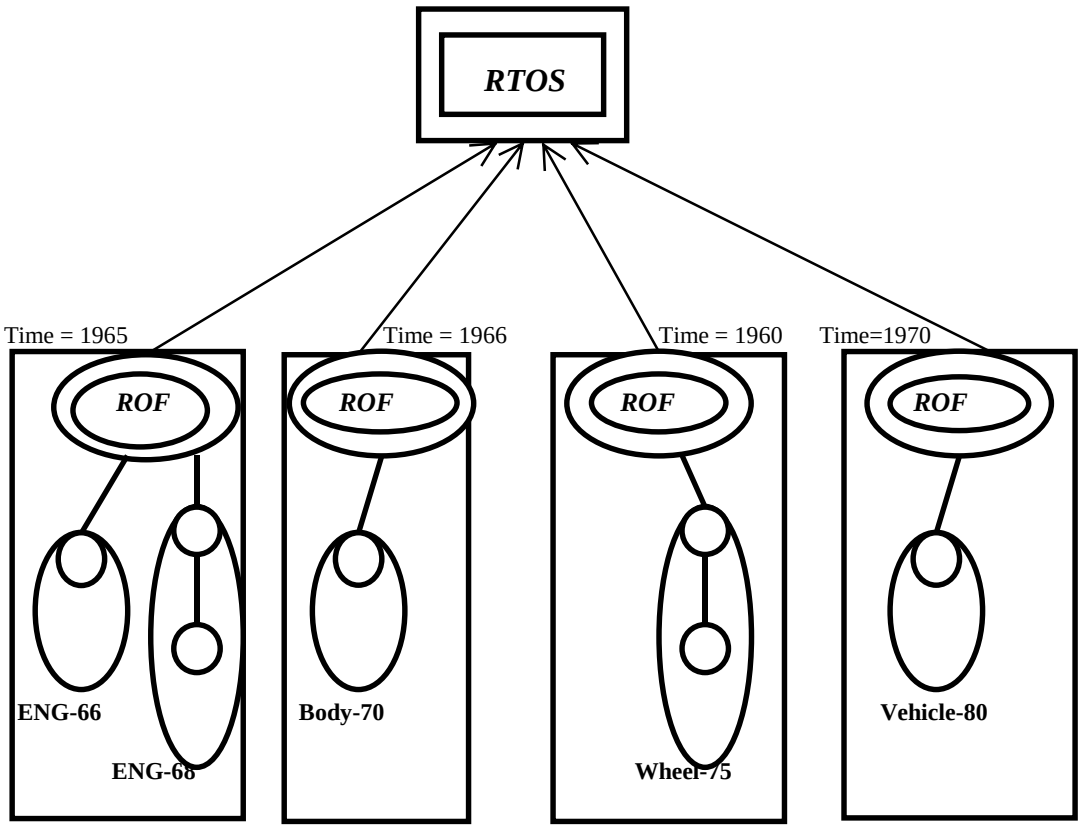


Figure 7: Vehicle design and development system - *an example* of TOS

In the class-based object-oriented systems, a class is used to assemble a set of objects which share some common assets as it is done in a family in the TOS. However, a family encapsulates more features than a class. For example, in a class, the structure of the class is always shared by all its states (or instances). A change to the class structure not only affects the states of the class, it also propagates to the structures and states of all subclass of the class. In a family, however, the structure or state of each temporal object of the family shares the *ROF* only at the time when its birth stage is created. After that, each temporal object is independent and a change in a temporal object does not effect the *ROF* or any other objects of the family. In other words, the *ROF* of a family is *read-only*, it does not change with time. Time is associated with a temporal object and *ROF* of the family (see Figures 7- 9).

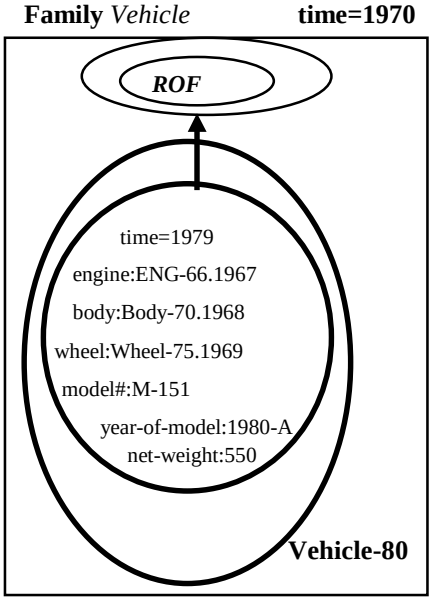


Figure 8: TCO *Vehicle-80* and it's subobjects

***ROF*(Vehicle)**
Aggregation-of:
{Engine,Body,Wheel**}**
Instance-Variables:
{ time=1979
model#,
year-of-model,
net-weight**}**
Methods:
{*assemble*, *test-it***}**

Figure 9: *ROF* of Family *Vehicle*

In the TOS two types of families, *simple families* and *complex families*, can be defined. A *simple family* represents an independent object development environment in which temporal objects can be constructed without sharing knowledge of other families. For example, in Figure 7, the families *Engine*, *Body* and *Wheel* are simple families. Two simple families do not share any knowledge. A simple family is analogous to a class in the class-based approach, which has no super-class like the *system class* or *root class*.

In existing class-based object-oriented systems, a complex object is defined as an object which can have another object as the value of a particular instance variable (Kim et al, 1987). We extend our definition of a family to *complex family* which provides a facility for the integration of non-homogeneous temporal objects of different families in order to build another temporal object which is referred to as *temporal complex object (TCO).* The components of a TCO are temporal objects of non-homogeneous families, and the temporal objects which take part in the construction of a TCO are called *subobjects* (or *components*) of the TCO (for details see (Fotouhi et al, 1992b; Shah et al, 1993b).

A new TCO, TCO_c, can be defined in a family F_c at a given time instance $t_{1,c}$ with r number of subobjects of r different simple families. The birth stage of the TCO, TCO_c, may be created at time instance $t_{1,c}$ if the temporal condition $(t_{1,k} \leq t_{1,c})\ (t_{Fc} \leq t_{1,c})$ is true for all k such that $1 \leq k \leq r$, where t_{Fc} is the time instance when the complex family F_c was created, and $t_{1,k}$ is the time instance when birth stage of the k-th subobject was created. This temporal condition ensures that all temporal subobjects and the complex family exist *before* the existence of the TCO. Figure 8 shows the birth stage of the complex family *Vehicle* which is an aggregation of three simple families *Engine, Body* and *Wheel*. In this figure the TCO *Vehicle-80* is constructed at time instance *1979* (denoted by *Vehicle-80.1979*) if the temporal condition, $(ENG\text{-}66.1967 \leq Vehicle\text{-}80.1979) \wedge (Body\text{-}70.1968 \leq Vehicle\text{-}80.1979) \wedge (Wheel75.1969 \leq Vehicle\text{-}80.1979) \wedge (Vehicle\text{-}Family.1970 \leq Vehicle\text{-}80.1979)$ is true, where *ENG-66.1967*, *Body-70.1968* and *Wheel-75.1969* are the time instances when the subobjects *ENG-66*, *Body-70*, and *Wheel-75* are constructed in their families, respectively, and *Vehicle.1970* is the time instance when the family *Vehicle* is constructed in the TOS. A TCO has all temporal parameters, time-span, life-span, and life-sequence, like other temporal objects. *ROF* of the family *Vehicle* which was defined at a time instance *1979* is given in Figure 9.

Within the boundary of a simple family, we use the *offspring* technique and among the families we prefer the *copying* technique for knowledge sharing (Alashqur et al, 1989; Borning,

1986). The aggregation and integration of temporal objects into a TCO can generate certain conflicts and compatibility problems such as naming and scaling between a TCO and its subobjects. For example, Naming conflicts occur when two or more subobjects of a TCO contain instance variables or methods with the same name such as the instance variable *weight* which has been defined in subobjects *Engine* and *Body* as well as in the TCO *Vehicle-80*. We are currently investigating these issues.

In (Fotouhi et al, 1994), we proposed a temporal object-oriented query language (TOOL) for the TOS. The query Language TOOL is a superset of SQL. The proposed language can answer both the temporal and non-temporal queries on families (simple and complex). The TOOL uses a set of logical operators and a set of temporal operators of the SQL and TSQL, respectively (Clifford, 1982) (for more details and sample queries see (Fotouhi et al, 1994)).

3. Operators of the Temporal Object System (TOS)

In this section, we describe a set of operations (or operators) that are proposed for the creation of families, temporal objects, and stages in the TOS. There are two main sets of these operations. The two sets of operations deal with the different operations that are related to the simple family and its temporal objects, and the complex family and its temporal object, respectively. Both types of families (simple family and complex family) are defined in the TOS as children of the *RTOS* by designing *ROF* at a time instance (see Figures 1 and 7). The creation of a new family in a TOS by designing its *ROF*, adds a new object development environment to the TOS in which the new types of objects can be created. Addition of a stage either adds a new temporal object or offstage object to a family, or it updates an existing temporal object by appending the new stage to the life-sequence of the existing temporal object.

3.1 Simple Family Operations

In this section, we propose operations for defining simple families and simple objects. The following set of four operations are needed to design a simple family and its temporal objects.

create-family (F_i, t_{Fi}): This operation creates a new family F_i in the TOS as child of the *RTOS* at a time instance t_{Fi} by allocating space for a new child node *ROF* of the family F_i. The contents of the *ROF* and a stage are the same; both contain two compartments: structure and state. The following set of two operations, fills the common knowledge (structure, state, or both) in the *ROF* of family F_i at the time instance t_{Fi}. The syntax of the two operations is as follows:

def-ROF-structure *(F_i)*
def-ROF-state *(F_i)*

The first operation *def-ROF-structure (F_i)* stores common instance variables and methods of the family in the structure compartment of the *ROF* of family F_i. This operation provides common structural and behavioral capabilities to all future temporal objects that will be defined in the family. The operation *def-ROF-state (F_i)* stores data values for the instance variables (if any) in the state compartment of the *ROF* of family F_i.

The following operations construct simple temporal objects in a given simple family. The syntax of the operations is given and described as follows:

create-object *(F_i, TO_j, t_{TOj})* if $t_{TOj} \geq t_{Fi}$: This operation creates a temporal object TO_j in a simple family F_i at a time instance t_{TOj} by allocating space for the birth stage of the temporal object TO_j. The temporal object is created only if the temporal condition ($t_{TOj} \geq t_{Fi}$) is true where t_{Fi} is the time instance when the family F_i was created in the TOS.

The birth stage of the temporal object can be filled in with structure, state, or both by the following two operations. These operations will also be used whenever a change occurs to a temporal object, and a new stage is defined to incorporate the change to the temporal object.

assign-structure(F_i , TO_j , S_{TOj}): This operation allows the user to assign instance variables and methods to the stage S_{TOj} for the temporal object TO_j. This operation further invokes separately the following two operations for defining instance variables and methods in the stage.

assign-state(F_i, To_j, S_{TOj}): This operation enables the user to assign data values to locally defined and shared instance variables of the stage S_{TOj} of the temporal object TO_j.

The following operation creates a new stage in a temporal object.

create-stage (F_i, TO_j, t_k) if $t_k \geq t_{TOj}$: The operation creates a new stage at time instance t_k in the temporal object TO_j of a family F_i, and it adds the new stage to the life-sequence of the temporal object (see Section 2). The stage is created if the temporal condition $t_k \geq t_{TOj}$ is true where t_{TOj} is the time instance when the last stage was defined in the temporal object. The current stage and the previous current stage are temporally dependent on each other. The operation stores any kind of change made to the temporal object TO_j in the form of a new (current) stage. The current stage can be filled by the operations *assign-structure( )*and *assign-state( )* which have already been described earlier.

After introducing a set of operations for constructing simple families and simple temporal objects, we now describe operations to construct offstage objects and to renovate them. The renovation operation on an offstage object brings a fresh copy of one or more participant temporal objects. The fresh copy is shared by defining a new stage of the offstage object at a time instance. Details about renovation of offstage object can be seen in (Shah et al, 1993a and Fotouhi et al, 1994)

create-offstage-obj $(TO_j, \bigcup_{k=1}^{k=r} \{TO_k * t_k\}, t_{ij})$: This operation creates an offstage object TO_j in a family environment by sharing knowledge of r number of temporal objects of the family at a time instance t_{1j}. Before explaining the syntax of the operation, we first describe the terms and the index k which are used in the operation. These r number of temporal objects are referred to as *participant objects*. There are r time instances t_k, where $1 \leq k \leq r$ corresponds to r number of

participant objects. A time instance t_k indicates the time instance in the life-span of the participant object TO_k, and all stages between the time instance t_k up to the time instance t_{1k} (when the participant object was created) from the life-sequence of the participant object are shared by the offstage object TO_j. In other words, the knowledge (partial or complete life-sequence) of the k-th participant object TO_k which lies in the time interval $[t_{1k}, t_k]$ is shared by the offstage object TO_j at the time instance t_{1j}. For each k, there exists an integer m where m is the number of stages of the participant object TO_k, and t_k is the time instance which identifies the offstage $S_{i,k}$ from the life-sequence of the participant object, which is the i-th stage of the participant object TO_k. There can be four different cases based on the values of the time instance t_k to identify a partial or complete life-sequence for every participant object.

The temporally-ordered set $\boldsymbol{\tau}_k = \{t_{1k}, t_{2k}, t_{3k}, ..., ..., t_{jk}..., t_{nk}\}$ consists of temporal parameters of the life-sequence $\boldsymbol{L}_k$ of the participant object TO_k. The set $\boldsymbol{\tau}_k$ and the time instance *now* are displayed graphically in Figure 10 on the time line. As mentioned earlier the time instance *now* represents *present* time instance. We have assumed earlier that the participant object TO_k has n number of stages at the time instance t_k when the offstage object TO_j is being constructed. Therefore there will be n different points on the time line corresponding to each stage, or corresponding to each member of the set $\boldsymbol{\tau}_k$, and one point is for the time instance *now* to show the *present* time instance (see Figure 10). The point t_0 is showing the origin or starting time instance of the time line. It may be interpreted as the time instance when the family or the TOS's application was created. The time instance t_k is displayed below the time line and its position is shown by an upward arrow at different places on time line in each case (see Figure 10).

In Figure 10, four possible cases are identified where the time instance t_k can be positioned in the life-sequence of the participant object TO_k. A complete or partial life-sequence of the participant object TO_k, which lies in the time interval $[t_{1k}, t_k]$ is shared by the offstage

object TO_j. The t_{1k} is the time instance when the birth stage $S_{1,k}$ of participant object TO_k was constructed. The time instance t_k can take four different positions in the life-sequence of the participant object based on its numeric value. The description of the four cases is as follows:

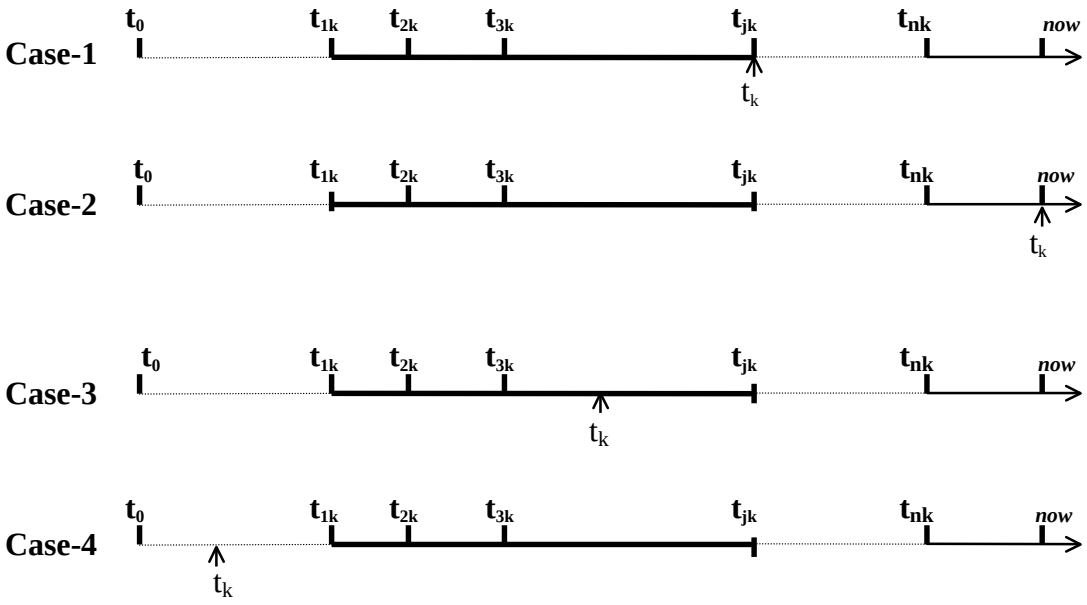


Figure 10: Four different possible cases of the time instance t_k

Case-1: $(t_k \in \tau_k) \wedge (t_k \neq now)$ In this case, the time instance t_k is a discrete time instance and it may lie at the beginning or at the end of time-span of a stage of the participant temporal object TO_k. In Figure 10/Case-1, time instance t_k is identified as a time instance t_{jk} which is a member of the set τ_k, i.e., $(t_k = t_{jk})$. It is the time instance when the j-th stage $S_{j,k}$ of the participant object is constructed. Therefore, the stage $S_{j,k}$ is the offstage, and a partial life-sequence $\{S_{1,k}, S_{2,k}, S_{3,k}, ..., S_{j,k}\}$ of the participant object TO_k will be shared by the offstage object TO_j.

Case-2*:* $(t_k = now)$, i.e., the time instance t_k is an extreme end of life-span $[t_{1k}, now]$ of the participant object TO_k (see Figure 10/Case-2). In this case, complete life-sequence of the participant object will be shared by the offstage object, because the time instance *now* includes the current stage of the participant object and its current stage is also the offstage.

Case-3: $(t_k \notin \boldsymbol{\tau}_k) \wedge (t_k \neq now)$, but the time instance $t_{jk} \in [t_{1k}, now]$, i.e., $t_{1k} \leq t_k \leq now$, where $[t_{1k}, now]$ is the life-span of the participant object TO_k. In this case, the time instance t_k is not a member of the set $\boldsymbol{\tau}_k$. The time instance t_k lies in time-span of any stage of the participant object, as shown in Figure 10/Case-3. In the figure, we have assumed that the time instance $t_k \in [t_{3k}, t_{4k}]$ which is the time-span of the stage $S_{3,k}$ of the participant object. This stage will be the offstage of the participant object and a partial life-sequence from the birth stage to the offstage will be shared by the offstage object TO_j. In Figure 10/Case-3, the stage $S_{3,k}$ is the offstage of the participant object TO_k and the partial life-sequence $\{S_{1,k}, S_{2,k}, S_{3,k}\}$ is shared by the offstage object TO_j at the time instance t_k.

Case-4: $t_k \notin [t_{1k}, now)$ It means that the time instance t_k does not belong to the set $\boldsymbol{\tau}_k$, the time instance t_k is a time point *before* the birth of the participant object TO_k, and its life-span does not include the time instance t_k. Therefore, no knowledge of the participant object will be shared in this case. In other words, the time instance t_k is not a valid time instance for the participant object, and no knowledge of the participant object can be shared by the offstage object TO_j at the time instance t_k.

The set $\bigcup_{k=1}^{k=r} \{TO_k *t_k\}$ enumerates a set of offstages from the set of participant objects that will be shared by the offstage object. In other words, a set of life-sequences (partial and complete) of the participant objects is extracted depending upon the values of the time instance t_k from every life-sequence of the participant object. The set of partial or complete life-sequences will be shared by the offstage object TO_j at the time instance t_k.

Before creating the birth stage of the offstage object TO_j at the time instance t_{1j} a temporal condition $(t_{1j} \geq t_k)$ for all $1 \leq k \leq r$ is verified. The birth stage $S_{1,j}$ of the offstage object TO_j shares knowledge from the finite set of participant objects. The set of offstage $\cup\{S_{i,k}\}$ where $1 \leq i \leq m$ and $1 \leq k \leq r$ and for each k $\exists$ an integer m such that $1 \leq i \leq m$, is

stored in the parents compartment of the birth stage $S_{1,j}$ of the offstage object. The birth stage of the offstage object is a complex stage in nature. Its other three compartments-structure, state, or both can be filled by using the set of the two operations (i.e., ***assign-structure ( )*** and ***assign-state( )***) which are already described.

ren-offstage-obj $(F_c,\ TO_c, \bigcup_{k=1}^{k=r} \{F_j * TO_j * t_j\},\ t_{TOc})$ if $t_{TOc} \geq t_j$ for all $1 \leq j \leq k \leq r$: This operation renovates an existing offstage object TO_c at the time instance t_{TOc} by creating its new stage by using a finite set of the participant object $\bigcup_{k=1}^{k=r} \{TO_k\}$. The semantics of the finite set $\bigcup_{j=1}^{j=r} \{F_j * TO_j * t_j\}$ and the time instance t_j are the same as described in the previous operation, but the range of the index j can be different, because the number of participant object taking part in the renovation may be equal to or less than the number of participant temporal objects which have taken part in the construction of the offstage object in the past where $past \leq t_{TOc}$. It is also possible that the number of stages of each participant object in the case of renovation may be greater than the number of stages of each participant temporal object which took part initially in the construction of the offstage object. The offstage object in renovated *after* validating the following two conditions:

(i) In the *past* where $(t_{TOc} \geq past)$ the temporal object TO_c was defined in a family as an offstage object by sharing a finite set of participant objects $\bigcup_{j=1}^{j=r} \{TO_j\}$. It means that the set $\bigcup_{j=1}^{j=r} \{TO_j\}$ of participant objects was also shared earlier at the time instance $t_{1,c}$ when the offstage object TO_c was defined by constructing its birth stage $S_{1,j}$. Now only those temporal objects which are member of the set $\bigcup_{j=1}^{j=r} \{TO_j\}$ can take part in the renovation of the offstage object TO_j at the time instance t_{TOc} where $(t_{TOc} \geq past)$. The time instance *past* is a time instance when the offstage object was constructed or renovated. The operation checks the membership of the participant

objects which are participating in the renovation. Because only those participant objects can participate in renovation of an offstage object which earlier participated in the construction of the offstage object. The alignments of the temporal objects may be different in the syntax of the creation and renovation operations of an offstage object.

(ii) The temporal condition $(t_{TOc} \geq t_j)$ must be true, where $1 \leq j \leq r$. This temporal condition ensures the temporal availability of knowledge of each participant object. It means that the knowledge which is not acquired by a participant object at the time instance t_j, cannot be shared now by the offstage.

3.2 Complex Family Operators

During the definition of a complex family, we define a configuration to reuse the knowledge from a finite set of the existing families in another family (complex family) of the TOS. The set of families is referred to as *constituent families*, and the temporal objects of the constituent families are referred to as *subobjects* of a temporal complex object (TCO).

In this section we describe a set of operations for designing a complex family, a TCO, and to renovate a TCO. The set of operations is described as follows:

create-comp-family $(F_c, \bigcup_{j=1}^{j=r} \{F_j\}, t_{Fc})$ if $t_{Fc} \geq t_{Fj}$: This operation creates a complex family F_c at the time instance t_{Fc} in the TOS as an aggregation of a finite set of families $\bigcup_{j=1}^{j=r} \{F_j\}$ where r is an integer, and it can be referred to as the *degree of the aggregation* which is the number of constituent families of the new complex family F_c. The set $\bigcup_{j=1}^{j=r} \{F_j\}$ defines a permanent configuration of the complex family F_c and all temporal objects of the complex family can use the temporal objects only from the set of families $\bigcup_{j=1}^{j=r} \{F_j\}$ after the creation of the complex family. The temporal condition for all j, $(t_{Fc} \geq t_{Fj})$ where $1 \leq j \leq r$ must be true *before* the creation of the family. The temporal condition ensures the temporal correctness of the creation of

the complex family F_c. The temporal condition also ensures that all constituent families must exist *before* participating in the aggregation of a new complex family. As all families are directly children of the root node of the TOS (see Figure 1 and Figure 7), so the root of the TOS (i.e., *RTOS*) is also the parent of the newly defined complex family F_c. Each family is created with the creation of its *ROF*. This operation creates a space for *ROF* of the family F_c and the set $\bigcup_{j=1}^{j=r} \{F_j\}$of families in its parents compartment. Later, the common knowledge of the complex family can be filled in by the two operations, i.e., ***def-ROF-structure*** *( )* and ***def-ROF-state*** *( ).* These operations are already defined in the previous section. After the operations *create-family* and *create-comp-family* which create a simple family and a complex family, respectively, the process of filling in the *ROF* of a family (simple or complex) is the same.

After describing the operations related to complex families, we now describe an operation to create a temporal complex object (TCO) in a complex family.

create-TCO $(F_c, TO_c, \bigcup_{j=1}^{j=r} \{F_j * TO_j * t_j\}, t_{TOc})$*:* This operation creates a TCO TO_c at a time instance t_{TOc} in a complex family F_c by sharing r number of subobjects of the families which are mentioned in the operation by the set $\bigcup_{j=1}^{j=r} \{F_j * TO_j * t_j\}$. The complex object is created if the temporal condition $(t_j \leq t_{TOc}) \wedge (t_{Fc} \leq t_{TOc})$ is true for all j, $1 \leq j \leq r$ where t_{Fc} is the time instance when the complex family was created. The temporal condition ensures that all temporal subobjects given in the operation and the complex family must exist *before* the construction of the TCO. The semantics of the set $\bigcup_{j=1}^{j=r} \{F_j * TO_j * t_j\}$ and the time instance t_j are the same as they are described in the operation for creating an offstage object. In the construction of a TCO in a complex family, a number of subobjects from different constituent families are transported into the family. The construction of a TCO causes *transportability* of knowledge among the families of the TOS. The birth stage of the TCO contains the set of r temporal subobjects. The

knowledge of r temporal subobjects from their respective families, is shared by the TCO through its birth stage $S_{1,k}$. The alignment of the member tuples (family, subobject, time instance) as given in the set $\bigcup_{j=1}^{j=r} \{F_j * TO_j * t_j\}$ may be different from the alignment of the set of families $\bigcup_{j=1}^{j=r} \{F_j\}$ that are used while designing the complex family F_c. The operation will be valid only if a family which is a member of the set $\bigcup_{j=1}^{j=r} \{F_j * TO_j * t_j\}$ is also a member of the set $\bigcup_{j=1}^{j=r} \{F_j\}$.

The birth stage of the TCO can be defined by sharing the local properties and/or assigning state using the two operations which are already defined for this purpose.

ren-TCO $(F_c, TO_c, \bigcup_{j=1}^{j=p} \{F_j * TO_j * t_j\}, t_{TOc})$: This operation renovates an existing TCO TO_c at a time instance t_{TOc}. The operation validates the following conditions before starting the process of renovation.

(i) The temporal condition ($t_j \leq t_{TOc}$) for all $1 \leq j \leq r$ must be true,

(ii) All temporal subobjects $\bigcup_{j=1}^{j=p} \{F_j * TO_j\}$ which are participating in the renovation must belong to the constituent families of the complex family F, and the subobjects which participated in the construction of the original TCO TO_c, can participate in the renovation of the TCO. The semantics for alignment of this operation are similar to those that are already mentioned in the create operation of a TCO.

The operation renovates the TCO TO_c by adding a stage, say, $S_{i,c}$ in the life-sequence of the TCO. The renovation brings a fresh knowledge of one or more subobjects in the TCO.

4. Implementing the Operators of the TOS

Figure 11 depicts the overall architecture of the temporal object system (TOS). Object Manager (OM) is an important module of the TOS and is responsible for providing the basic functionality to the TOS. We developed a prototype of the OM that performs a subset of the set

of operations of the OM, that are described in Section 4. The developed OM performs three basic operations: creation of a new temporal object in a simple family, creation of a new stage in an existing temporal object, and creation of an offstage in a family. The operations concerning renovation of a complex temporal object and offstage object and other operations related to the creation of complex families and complex temporal objects (see Section 4) are not yet implemented. We are actively working on these operations and in the near future they will be added to the developed OM. A complete design of the OM was reported in (Sayegh et al, 1994). In the next section we give salient features of the design.

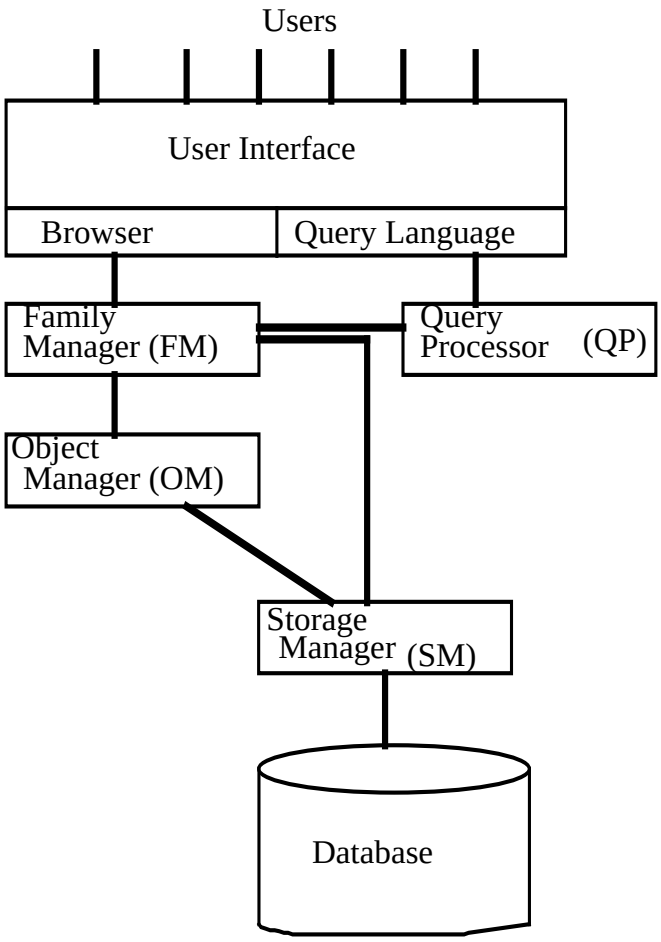


Figure 11: Architecture of the Temporal Object System (TOS)

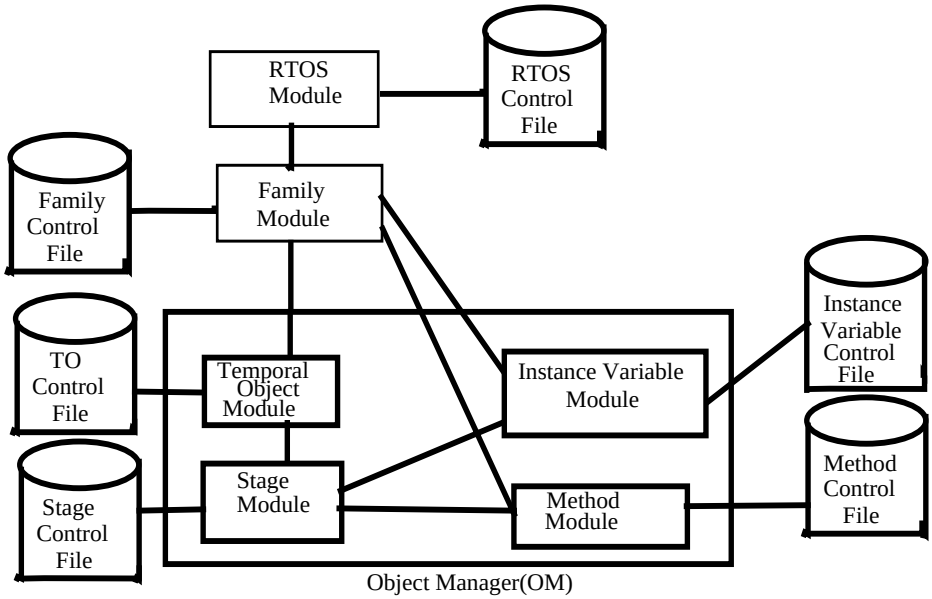


Figure 12: Architecture of the object manager and other modules

4.1 Design of the Modules

The OM consists of four modules: *temporal object module*, *stage module*, *instance-variable module*, and *method module* (see Figure 12). As shown in the figure, Family Module invokes the sub-modules of the OM to retrieve and store information such as *ROF* of a family. Family Module is invoked by the main driver, i.e., RTOS Module (see Figure 12). In the next three sections we describe the four sub-modules (or components) the OM.

The architecture of the object manager and other modules is depicted in Figure 12. The RTOS Module interacts with *Family Module*, and *Family Module* interacts with *Temporal Object Module* (see Figure 12). These two modules, i.e., *Family Module* and *RTOS Module*, are not components of the OM (the components of the OM are shown in a rectangle in Figure 12) but they are also developed because their development is necessary for the working of the OM module. Here we briefly describe their functions and workings of these modules.

The module *RTOS* loads the RTOS-Control-File (file layout is given in the next section) once the system is turned on. The basic function of this module is to load the system primitives and to accept the commands. This module interacts with Family Module for system-level browsing. When a new family is defined in the system, the module *RTOS* validates the temporal condition (for details see Section 4) before the creation of the new family.

Family Module is responsible for the creation of new families and their *ROF*s. It stores and retrieves *ROF*s of the newly defined families in Family-Control-File (layout is given in the next section). This module also validates the temporal condition before the creation of a new temporal object. This also interacts with *Instance-Variable Module* and the *Method Module* of the OM while defining the instance-variables and the methods in the *ROF* of the family, respectively. We summarize the main functions of the Family Module as follows:

(i) It stores and retrieves *ROF* and other information of a family.

(ii) It keeps track of the instance-variables and methods associated with *ROF* of a family.

(iii) It keeps track of temporal objects associated to a family.

4.1.1 Temporal-Object Module

The main task of this module is to create a new temporal object in a family, and to invoke the *Stage Module* that creates the birth stage of the new temporal object or creates a current stage of an existing temporal object. This module also validates the temporal condition before the creation of a temporal object. The *Temporal-Object Module* is mainly responsible for storing and retrieving the information pertaining to temporal objects from the Temporal-Object-Control-File (layout is given the next section), and to keep track of association between temporal objects and their stages.

4.1.2 Stage Module

This module gives an environment for creating a new stage in a temporal object and fill-in the contents (instance-variables and methods) of the stage by activating the two modules: Instance-Variables Module and Method Module (see Figure 12). The module is responsible for the storage and retrieval of the contents of stages in Stage-Control-File. keeps track of the associations between stages and its instance-variable and methods .

4.1.3 Instance-Variables and Method Modules

The Instance-Variable module basically stores and retrieves the data type and data of instance-Variables from Instance-Variables-Control-File (layout is given in the nest section).

This module also keeps track of association between instance-variables and stages of a temporal object.

The Method Module stores and retrieves methods from Method-Control-File (layout is given in the next section). This module is also responsible for the compilation, code generation, and invocation of the methods. This module keeps track of association between the methods and stages of a temporal object.

4.1.4 Control Files

The four modules of the Object Manager, Family Module, and RTOS Module that are described in the previous sections. These modules use five different control files for storage and retrieval purposes. The layouts of these control files are briefly given as follows:

RTOS-Control-File: A file contains information about the application such as application creation time and name of application.

Family-Control-File: This file keeps information about the families of the application. It contains family creation time, family name, starting address in Temporal-Object-Control-File of the first temporal object in a family, number of temporal objects in the family, starting address of the first instance-variable of *ROF* of the family in Instance-Variable-Control-File, number of instance-variables in *ROF* of the family, starting address of the first method record of ROF in the family in the Method-Control-File, and number of methods.

Temporal-Object-Control-File: This contains temporal object creation time, system generated Object Identification Number (*OID*), temporal object name, starting address of the birth stage record in the Stage-Control-File, and number of stages in the temporal object.

Stage-Control-File: This file contains stage creation time, starting address of the first instance-variable of stage in Instance-Variable-Control-File, number of instance-variables in the stage, starting address of the first method record in the Method-Control-File, and number of methods in the stage.

Instance-Variables-Control-File: This file contains instance-variable name, instance-variable type, and data value of instance-variable (if any).

Method-Control-File: This file contains method name, and method internal name (the internal name is system generated). The reason to keep this name is to avoid the conflict in the method names if the user creates two methods with the same name.

4.2 Operators Implementation Using SELF

4.2.1 Hardware and Software Configuration

We have implemented the object manager using the programming language SELF version 4.0 on a SUN Sparc Workstation running Solaries 2.4. The programming language SELF is a prototype-based object-oriented programming (OOP) language that is originally developed at Stanford University, and experimental versions of its compiler are developed by SUN Corporation (Chambers et al, 1989).

4.2.2 Benefits of Using SELF for the Implementation

We have selected this language as the implementation tool to implement the three main modules which are the three groups of the TOS operators for a number of reasons (see Figure 12). The SELF is a prototype-based and graphical-oriented OOP language. As mentioned before the TOS model is a hybrid of the prototype-based and the class-based approaches and the SELF makes the contents of each stage whether it belongs to a temporal object or *ROF* of a family transparent to the user. This transparent property of the SELF makes all the changes (stages) to a temporal object visible to the user and he can glance through the evolution of the temporal object The SELF supports the concepts of both instance-variables and methods. Any entry in an object in SELF is called a *slot* (Chamber et al, 1989). The slot could be a data slot or code slot, each has its own behavior. Data slots return the data they have once *called*. Code slots, on the other hand, perform a task once they receive a message invoking them. Code slots (SELF's understanding of methods) can have parameters sent to them and a receiver object to send the results to. The language to code these slots are a SELF variation of SmallTalk.

By using SELF capabilities, custom objects can be created very easily. In terms of objects' creation, the operators in the Object Manager of the TOS need a system that allows creating families, temporal objects, and stages. SELF allows the end-user to create whatsoever object he/she thinks of. We have created all the objects the TOS requires in addition to an object to be a reference to all objects, which is the RTOS object that represents the 'root' of the system and will hold the families within.

In addition to creating objects, SELF provides the facility to create *sequences.* Sequences are objects in SELF that contains information about a list or group of objects. Using sequences we were able to group families under RTOS together, temporal objects together under any familyr, and stages under a temporal object in one bag.

4.2.3 SELF Implementation of the Modules

Our implementation of the TOS operators on SELF included the main system object (RTOS, simple family, temporal object, and stage) with all their system-oriented attributes that were mentioned as fields in the control files (see Section 4.4). The fields of the control files are implemented as slots of objects, for example, fields of the RTOS-Control-File are implemented as slots in the RTOS object. For the temporal condition validation methods, we have added code-slots to the RTOS object to create families, temporal objects, and stages after validating the temporal condition provided as parameters to these methods. These methods can be accessed from anywhere in the TOS hierarchy since they are in the RTOS which is seen by all objects in the TOS. The object manager of the TOS allows the user to create and maintain any number of temporal objects in a family, each with any number of attributes, methods and slots.The only limit is the amount of free memory available to the SELF 4.0 system. The TOS maintains some internal objects such as *RTOS* (the root object), and it keeps some objects to maintain the Window environment.

4.2.3.1 The RTOS Object

The *RTOS* object contains instance-variables and methods necessary for maintaining the TOS hierarchy with its various objects. It also maintains a list of all families in the system through **familiesList** (see Figure 13). Further it stores the name of the application and points to a time instance objects which records the creation time of the application as a whole.

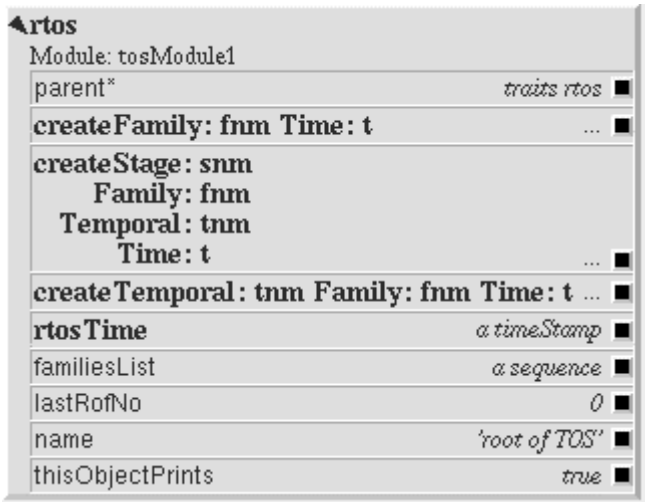


Figure 13: The 'RTOS' object and its slots

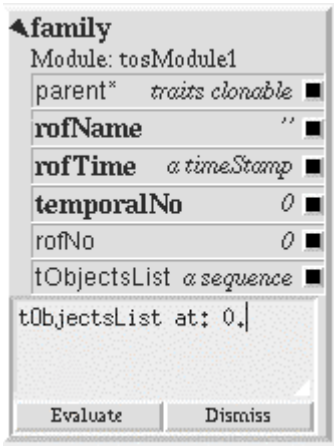


Figure 14: The 'family' object

Since *RTOS* object is considered to be a unique object within the TOS, it was defined as a descendant of '*traits oddball*' which in SELF 4.0, is considered to be a prototype from all unique non-colorable objects. This root object is automatically displayed whenever the system is initiated. Among other slots, the RTOS contains the necessary methods for creating new instances of families, temporal objects and stages, namely: the **CreateFamily, CreateStage** and **CreateTemporal** methods.

4.2.3.2 The 'Family' Object - Family Module

The 'Family' object is intended to be a *prototype* of all families that will be created in the system. A 'family' is declared to be a descendent of *traits clonable*, such objects can be cloned whenever necessary. The family object also keeps tracks of the family's name, time of creation, number of temporal objects within, and a list of the enclosed temporal objects. In addition, other

attributes and methods can be added dynamically at run time to meet the requirements of the problem at hand.

4.2.3.3 The 'Temporal' Object - Object Module

The temporal object is intended to act as a *prototype* for the TOS's temporal objects that are meant to be created and enclosed within families. A family may have any number of temporal objects as long as there is enough system memory.

The temporal object is also a descendent of *traits clonable.* It stores the time of its creation, the number of stages within, its ID number, as well as the number and list of enclosed stages. Other slots can be added and manipulated dynamically at run time.

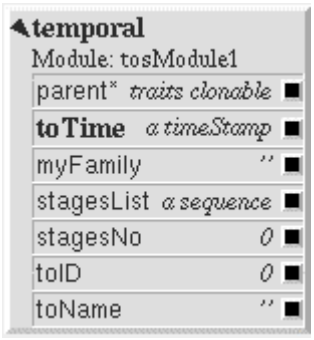


Figure 15: The 'Temporal' object

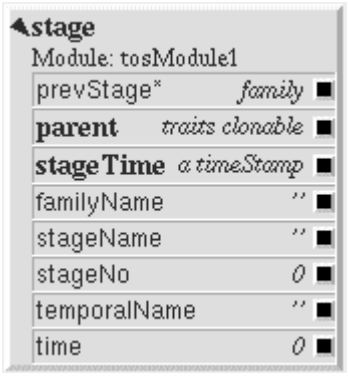


Figure 16: The 'Stage' object

4.2.3.4 The 'Stage' Object

Within a temporal object, there is a list of stages. The stages represent entities and changes applied on them as time evolves.

The stage object acts as a *prototype* for creating new stages. The stage object records the stage creation time, the name of the family it belongs to, the individual stage number, the name of the temporal object it belongs to. Additionally, to enable a stage to inherit methods and attributes from preceding stages, a **prevStage** slot is declared. This slot is intended to point to the immediate preceding stage and to inherent its attributes and methods.

4.2.3.5 The ‘tosWorld’ Object

The ‘tosWorld’ object is used to maintain the TOS World environment along with its user interface elements and labels. The ‘tosWorld’ object implements two useful methods:

*** ‘init’** for initiating and displaying a new instance of the TOS World Window, which is the main interface to the Object Manager, and

- **‘kill’** to terminate a TOS World Window.

4.2.3.6 The Time Stamp Object

The ‘timestamp’ object is a general object used by the different objects within the TOS to record the creation time. If a programmer likes to change time format and/or intervals he only needs to tamper with the ‘timestamp’ object.

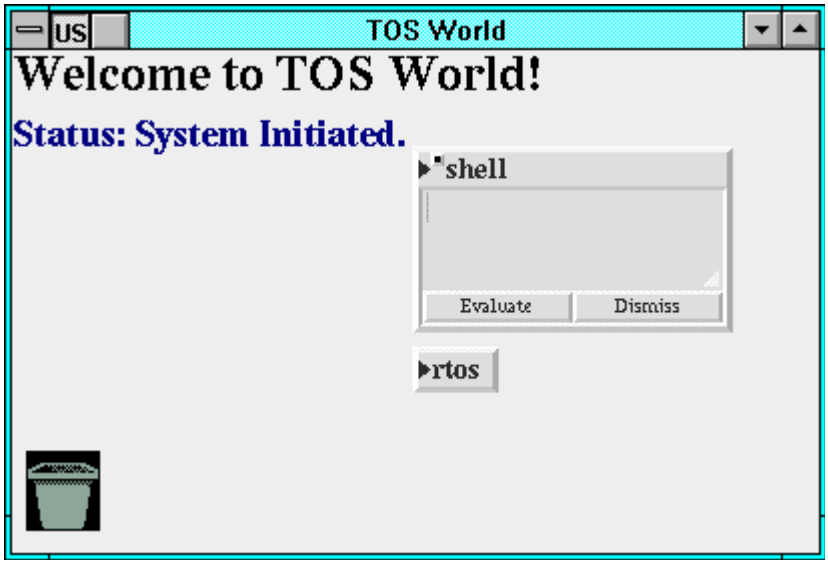


Figure 17: The 'tosWorld' Window

4.3 Using the Object Manager

There are two ways for the end-user to utilize the OM in the environment of SELF. A user can directly use the graphical interface of the ‘tosWorld’ that is built over the graphical interface of SELF 4.0. In this way, the user can create and manipulate objects visually. The whole ‘tosWorld’ environment along with its various families, stages and temporal objects can be preserved by the ‘save snapshot’ feature of SELF 4.0. As an alternative, a user may write a

complete SELF script that utilizes the functionality made available within the 'RTOS' object. Such scripts can be run under the 'tosWorld' environment directly.

5. Conclusions and Future Work

5. 1 Conclusions

We briefly described the TOS, an object-oriented system, which uses a hybrid approach of the class-based and prototype-based approaches, and utilized the merits of both techniques. The concept of family is similar to the concept of a class, with the difference that families in the TOS are not organized into a class-hierarchy (or class lattice) as it is done in the class-based object-oriented systems. The TOS is a collection of families organized in a two-level hierarchy. We extended the concept of a family to a complex family in order to share knowledge of existing families. The concept of complex family enhances reusability and transportation of objects knowledge among them. We associate time dimension with each activity of the TOS.

In this paper we defined and described the operators of the TOS and reported on their implementation. These operators are grouped into three different modules: Object Module (or Object Manager), RTOS Module, and Family Module The grouping is based on the functionality's of the operators. The operators are implemented using OOP SELF version 4.0. This implementation gives a basic environment of the TOS where simple families and simple temporal objects can be created and updated.

5.2 Future Work

We intend to further extend the current implementation of modules to include remaining un-implemented operators of these modules. This extension will provide more functionality to the TOS such as creation of complex families, temporal complex objects, and renovation of objects. We also intend to develop other modules of the TOS that are given in Figure 11.

Additionally, we are in progress of porting TOS to Java. The project will focus on providing a graphical platform-independent system in which the object manager will run as a

Java applet with a GUI interface such that it can be executed within a web browser while providing the ability to create and manipulate objects graphically. Furthermore, by using emerging technologies like Java's support for RMI *(Remote Method Invocation)*, we are considering to expand the functionality of the OM so it can support the creation and manipulation of distributed objects.

References

Agrawal, R., Buroff, S., Gehani, N., and Shasha, D (1991), "Object Versioning in Ode," *Proceedings of the 7th IEEE Int. Conference on Data Engineering*, pp. 446-445.

Alashqur, A. M., Su, S. Y. W., and Lam, H. (1989), "OQL: A Query Language For Manipulating Object-Oriented Databases," *Proceedings of 15th International Conference. on VLDB*, pp. 433-442.

Borning, A. H. (1986), " Classes Versus Prototypes in Object-Oriented Languages," *ACM/IEEE Fall Joint Conference*, pp. 36-40.

Chambers, C., Ungar, D., and Lee, E. (1989), "An Efficient Implementation of SELF a Dynamically-Typed Object-Oriented Language Based on Prototypes," *Proceedings of the ACM International Conference on Object-Oriented Programming Languages, Systems and Applications (OOPSLA'89)*, pp. 49-70.

Clifford, J. (1982), "A Model for Historical Databases," P*roceedings of Logical Bases For Databases*, Toulouse, France.

Dutta, S., (1989), "Generalized Events In Temporal Databases," *Proceedings of the 5th IEEE International Conference on Data Engineering*, pp. 118-126.

Fotouhi, F., Shah A., and Grosky, W. (1992a), "TOS: A Temporal Object System," *Proceedings of the 4th Inl. Conference on Computing. & Information, Toronto, Canada, pp. 356.*

Fotouhi, F., Shah A., and Grosky, W. (1992b), "Complex Objects in the Temporal Object System," *IEEE Post-Proceedings of the 4-th Int. Conference on Computing & Information*, pp. 384.

Fotouhi, F., Shah. A., Ahmed, I., and Grosky (1994), TOS: A Temporal Object-Oriented System," *Journal of Database Management, 5(4),* pp. 3-14.

Gadia, S. K., and Yeung, C. S. (1991), "Inadequacy of Interval Timestamps in Temporal Databases," *the Information Sciences Journal, 541(1&2),* pp. 1-22.

Katz, R. H., Chang, E., and Bhatega, R. (1986), "Version Modeling Concepts for Computer-Aided Design Databases," *Proceedings of the ACM SIGMOD Conference*, pp. 379-386.

Kim, W., Banerjee, J., et al (1987), "Composite Object Support in an Object-Oriented Database System," *Proceedings of the ACM International Conference on Object-Oriented Programming Languages, Systems and Applications (OOPSLA'87),),* pp. 118-125.

Kim, W., (1990), "Introduction to Object-Oriented Databases," The MIT Press, Cambridge, Massachusetts.

Ling D. H. O., and Bell, D. A. (1990), "Taxonomy of Time Models in Databases," *Information & Software Technology Journal, 32(3),* pp. 215-224.

Maier, D. (1986), "Why Object-Oriented Database Can Succeed Where Others Have Failed," *Proceedings of International Workshop on Object-Oriented Database Systems*, pp. 227.

Nguyen, G. T., and Rieu, D. (1989), "Schema Evolution in Object-Oriented Database Systems," *Data & Knowledge Engineering Journal, North-Holland, 4(1),* pp. 43-67.

Roddick, J. F., (1992), "Schema Evolution in Database Systems - An Annotated Bibliography," *ACM-SIGMOD Record, 21(4*), pp. 35-40.

Sayegh, T., Shah, A., Faraj, I., Fotouhi, F., and Grosky, W. (1994), "Design of Object Manager For the TOS," *the VIII International Symposium in Informatics Applications (INFONOR94),* Chile, pp. 103-109.

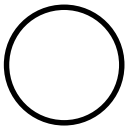
Shah, A., (1992), "TOS: A Temporal Object System," *Ph.D. Dissertation*, Wayne State University, Detroit, Michigan, USA.

Shah, A., Fotouhi F., and Grosky, W. (1993a), "Offstage objects and their Renovation in the Temporal Object System TOS," *the 3rd International Symposium on Database for Advance Applications*, Daejeon, Korea, April 6-8, pp. 306-312.

Shah, A., Fotouhi, F., and Grosky W. (1993b), "Renovation of Complex Object in the TOS," *the 12th Annual IEEE International Phoenix Conference on Computers & Communications*, Scottsdale, Arizona, USA, pp. 203-209.

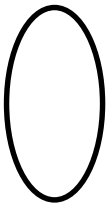
Shah, A., Fotouhi, F., Grosky, W., Al-Dhelaan, A., and Vashishta, A. (1993c), "A Temporal Object System For a Construction Environment," *Proceedings of the XIII Conference of the Brazilian Computer Society (SEHOSH-90), Vol. 1,* pp. 211-225.

Appendix

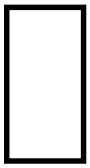


A circle represents a stage of a temporal object.

Time

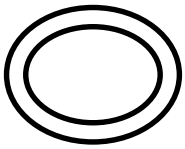


An oval shape represents a temporal object.

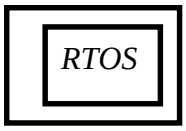


A rectangle shape represents a family.

Time



A double oval shape represents a root-of-family (ROF).



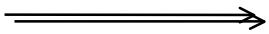
A double rectangle shape represents root of the *TOS- RTOS*.

(1)

A single point arrow represents direction of next stage of a temporal object.

(2) 

A dark arrow represents temporal inheritance.

(3) 

A double arrow represents Renovation or temporal repeated inheritance.