ODMG language extensions for generalized schema versioning support

Fabio Grandi and Federica Mandreoli

C.S.I.TE.-C.N.R. – D.E.I.S.
University of Bologna, Viale Risorgimento 2, I-40136 Bologna, Italy
`<fgrandi,fmandreoli>@deis.unibo.it`

Abstract. The management of different schema versions is required in long-lived database systems to accomplish data structural changes and represent their history. Once a suitable data model for schema versioning support has been defined, appropriate extensions must also be introduced in the data definition and manipulation languages. Such an extension is aimed at making the versioning facilities available at user-interface level and is the basis for the development of advanced multi-schema applications. In this paper we present extensions to the definition and manipulation language of the standard object-oriented data model ODMG for a generalized schema versioning support. To this end, two versioning modalities will be considered in a single powerful system: temporal versioning and management of alternative design versions. As far as the temporal components are concerned, the proposed extensions of ODL and OQL will be consistent with the TSQL2 temporal query language.

1 Introduction

Databases and software systems have complex structures which are likely to continually undergo changes during their lifetime. This is certainly true for OODBs, which have been mainly developed to model highly dynamic application scenarios where not only the data, but also their structure (i.e. schema) is subject to change. The need for retaining data entered under any schema definition has led to the introduction of the *schema version* notion. Generally speaking, to manage and maintain versions of a schema means dealing with one of the possible representations of the structure of the modeled real world. The application realm of schema versions may range from the maintenance of legacy data (formatted according to past schemas) to the reuse of software components, from the planning of human activities against alternative scenarios to the management of complex design processes.

In the object-oriented field, two kinds of schema versions have been considered: alternative versions (branching approach) and temporal versions. The former is typical of design environments (CAD/CAM applications), whereas the latter is required to model histories of structural changes (more suitable for GIS and multimedia applications). The first model which integrates the two versioning modalities, in order to enhance the expressive power and application potentialities of a single OODBMS, is presented in [6]. The model is based on an ODMG Release 2.0 [2] extension. The standardization infrastructure offered by the adoption of ODMG as a stepping stone is directed towards a high degree of portability and interoperability between systems and represents a wide endorsement of the object-oriented approach. The ODMG standard includes three major components: the *Object Model*, the *Object Definition Language* (ODL) and the *Object Query Language* (OQL). While [6] represents a desired extension at object-model level, the addition of generalized schema versioning support to the ODMG standard still needs an intervention at language level. Such an extension of ODL and OQL is the main purpose of this work.

The paper is organized as follows: Section 2 presents a brief overview of the model supporting generalized schema versioning; in Section 3 we propose our extension to the ODMG languages to make all the model facilities accessible to users with an SQL-like interface; conclusions will be found in Section 4.

2 Outlook of the underlying model

The object data model we consider here is a general model for the management of versions which integrates the pure temporal schema versioning with the branching versioning approach [6]. It has been developed in the framework of a comprehensive research project, in which the authors play an active part and to which [4–6] and the present work are all contributions. In this generalized model, the identification of a particular schema version relies on the use of a symbolic name (to denote a design alternative) and two time coordinates (to select a temporal version with respect to transaction and valid time [7]). Therefore, a multidimensional mechanism is needed to reference distinct versions also at language level. Any version could be referenced through a *bitemporal pertinence* and/or a user-defined name (*label*). A bitemporal pertinence is defined as a disjoint union of rectangles, where each rectangle is the product of a transaction- and valid-time interval (in accordance with the BCDM model [8]). All versions having the same symbolic label share a common property since, for example, they belong to the same consolidated version in engineering activities or the same scenario in GIS. At model level, a database can be represented through a directed acyclic graph (DAG), corresponding to the version derivation hierarchy used for CAD/CAM applications. Versions having the same label belong to the same *node* in the database graph, whereas the relationships between different nodes (e.g. the derivation of a new node or the merger of one node into another) are modelled through the graph *edges*. All the DAG elements are timestamped with transaction time in order to keep track of all the schema changes effected in the system.

Schema changes are supported by means of a collection of primitive algebraic operations. They are grouped into two sets on the basis of the DAG element on which they operate. The set of “schema changes on node” primitives includes a complete set of operations acting on the elements of the object-oriented data model supported, such as attributes and classes. The set of “schema changes on edge" primitives provides support for the integration of characteristics of a schema version with schema versions belonging to other nodes. For example, the primitive to merge versions (**MergeVersion** in [6]) creates a new schema version by merging schema versions (and their corresponding extant data) belonging to different nodes. A listing of all the primitives supported by the model and a definition of their formal semantics can be found in [6]. All the schema modification statements considered in this paper can always be implemented by means of such primitives, although we will not show details here, for the sake of brevity.

3 Extensions to the ODMG languages

The proposed ODMG language extensions follow some basic principles:

- they are SQL-compatible as OQL is very close to SQL 92;
- the ODL extensions support a complete set of primitive schema changes. In particular, they support all semantic constructs for the general schema versioning mechanism presented in [6];
- the versioning granularity is the schema;
- an internal approach to schema changes is adopted;
- they are TSQL2-compatible [11] as far as temporal parts are concerned.

In the object-oriented field, an important aspect which involves the schema versioning is the choice of the level at which versioning is supported. The alternative is between the single class and the entire schema. We adopt the schema as versioning granularity since it automatically provides a complete view of the set of class versions tied together in a schema version. In this way, it makes it easier to check the inter-class consistency and the management of queries on objects belonging to different class versions.

Following the internal approach to schema changes, when the schema undergoes a change, the definition of a complete new version to be added is not allowed and the only way of doing it is to apply a sequence of primitive schema changes to an already existing version. In this way, we provide the system with default semantics for automatic database conversion and an automatic consistency checking associated with each schema change primitive [4]. A schema change primitive is a non-decomposable operation acting on the schema. The support of a complete set of primitive changes allows the execution of any possible schema update. In fact, complex schema updates can be effected via sequences of schema change primitives.

3.1 Schema selection

The schema version selection is achieved through the `SET SCHEMA` statement included as OQL extension. It is basically the same statement introduced for TSQL2 [3, 11], augmented with a `LABEL` clause. Its complete form is:

```
SET SCHEMA <schema selection condition>

<schema selection condition> ::=
    LABEL <label>
        AND VALID <datetime value expression>
            AND TRANSACTION <datetime value expression>
```

It allows default values to be set for label, valid and transaction time to be employed by subsequent statements. The value set is used as a default context until a new `SET SCHEMA` is executed or an end-of-transaction command is issued. Notice that, although the valid-time and label defaults are used to select schema versions for the execution of any statement, the transaction-time default is used only for retrievals. Owing to the definition of transaction time, only current schema versions (with transaction time equal to *now*) may undergo changes. Therefore, for the execution of schema updates, the transaction-time specified in the `TRANSACTION` clause of the `SET SCHEMA` statement is simply ignored, and the current transaction time *now* (see Subsection 3.5 for more details) is used instead. Moreover, one (or two) of the selection conditions may not be specified. Also in this case preexisting defaults are used.

In general, the `SET SCHEMA` statement could "hit" more than one schema version, for example when intervals of valid- or transaction-time are specified. To solve this problem we distinguish between two cases:

- for schema modifications we require that only one schema version is selected, otherwise the statement is rejected and the transaction aborted;
- for retrievals several schema versions can qualify for temporal selection at the same time. In this case, retrievals can be based on a completed schema [10], derived from all the selected schema versions.

Further details on multi-schema query processing can be found, for instance, in [3, 9]. The scope of a `SET SCHEMA` statement is, in any case, the transaction in which it is executed. Therefore, for transactions not containing any `SET SCHEMA` command, a global default should be defined for the database. As far as transaction and valid time are concerned, the current and present schema is implicitly assumed, whereas a global default label could be defined by the database administrator by means of the following command:

```
SET CURRENT_LABEL <label>
```

Obviously, this definition is overridden by explicit specification in `SET SCHEMA` statements found in transactions.

3.2 Extensions to the ODL for direct DAG manipulation

ODMG includes standard languages (ODL and OQL) for object-oriented databases. Using ODL, only one schema, the initial one, can be defined per database. The definition of the schema consists of specifications concerning model elements, like interfaces, classes and so on. The main object of our ODMG language extensions is to enable full schema development and maintenance by introducing the concept of schema version at language level.

Node creation statement

A schema version can only be explicitly introduced through the `CREATE SCHEMA` statement added to ODL. It requires a new label associated with the new schema version. In this way, the explicit creation of a schema version always coincides with the addition of a new node to the DAG. Afterwards, when the schema undergoes changes, any new schema version in the same node may only be the outcome of a sequence of schema changes applied to versions of the node, since we follow the internal change principle. The `CREATE SCHEMA` syntax is:

```
CREATE SCHEMA <label>
    <schema def>
[<schema change validity>]

<schema def>::=
    <specification> | FROM SCHEMA <schema selection condition>

<schema change validity> ::=
    VALID <datetime value expression>
```

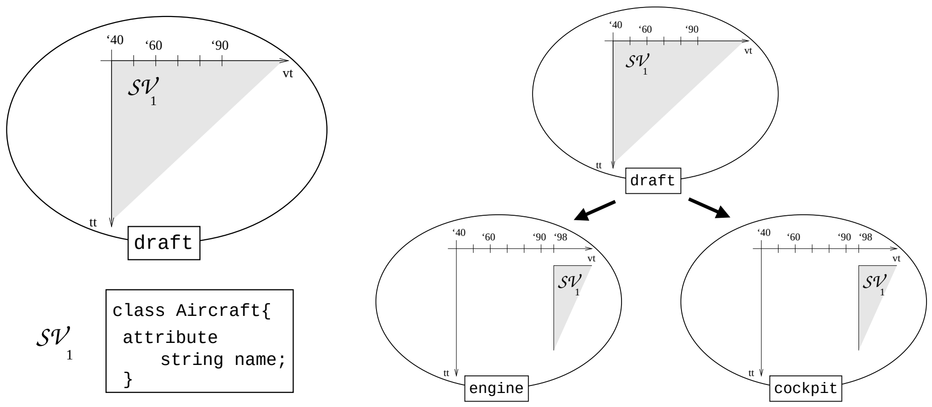
The `CREATE SCHEMA` statement provides for two creation options:

– the new schema version can be created from scratch, as specified in the <specification> part. In this case the new node is isolated;
– the new schema version can be the copy of the current schema version selected via the <schema selection condition>. In this case, the node with the label specified in the `FROM SCHEMA` clause becomes the father of the new node with the label specified in the `CREATE SCHEMA` clause.

The optional <schema change validity> clause is introduced in order to specify the validity of the schema change, enabling retro- and pro-active changes [4]. The new schema version is assigned the "version coordinates" <label>, $[now, \infty]$ and validity <datetime value expression>. When the <schema change validity> clause is not specified, the validity is assumed to be $[-\infty, \infty]$. The node creation is also recorded in the database DAG by means of a transaction timestamp $[now, \infty]$, which is associated with the node label.

Let us consider as an application example the activity of an engineering company interested in designing an aircraft. As a first step, an initial aircraft structure (schema) is drawn, whose instances are aircraft objects. This can be done by the introduction of a new node, named "draft", that include one schema version (valid from 1940 on) containing the class "Aircraft", as follows:

```
CREATE SCHEMA draft
    class Aircraft{
        attribute string name};
VALID PERIOD '[1940-01-01 - forever]'
```



```
class Aircraft{
 attribute
    string name;
 }
```

Fig. 1. Outcomes of the application of `CREATE SCHEMA` statements

The left side of Figure 1 shows the schema state after the execution of the above statement. In a second step, the design process can be entrusted to independent teams to separately develop the engines and the cockpit. For instance, the team working on the cockpit can use a new node, named “cockpit” derived from the node “draft” by means of the statement (the outcome is shown on the right side Figure 1):

```
CREATE SCHEMA cockpit
   FROM SCHEMA LABEL draft AND VALID AT DATE 'now'
VALID PERIOD '[1998-01-01 - forever]'
```

The new node contains a schema version which is a copy of the current schema version belonging to the node “draft” and valid at “now” ($\mathcal{SV}_1$). The temporal pertinence to be assigned to the new schema version is $[now, \infty] \times [1998/01/01, \infty]$. In a similar way, a new node “engine” can be derived from the node “draft”.

Node deletion statement

The deletion of a node is accomplished through the `DROP NODE` statement. Notice that the deletion implies the removal of the node from the *current* database. This is effected by setting to “now” the end of the transaction-time pertinence of all the schema versions belonging to that node and of the node itself in the DAG. The syntax of the `DROP NODE` statement is simply:

```
DROP NODE <label>
```

The `DROP NODE` statement corresponds to a primitive schema change which is only devoted to the deletion of nodes. Thus, the node to be deleted has to be isolated. The isolation of a node is accomplished by making the required `DROP EDGE` statements precede the `DROP NODE` statement.

Edge manipulation statements

Edges between nodes can be explicitly added or removed by applying the `CREATE EDGE` or `DROP EDGE` specifications, respectively. The corresponding syntax is:

`CREATE EDGE FROM` <label> `TO` <label>

`DROP EDGE FROM` <label> `TO` <label>

The `CREATE EDGE` statement adds a new edge to the DAG with transaction-time pertinence equal to $[now, \infty]$. The `DROP EDGE` statement removes the edge from the current database DAG by setting its transaction-time endpoint to *now*.

3.3 Extension to the ODL for schema version modification

Schema version modifications are handled by two collections of statements. The former acts on the elements of the ODMG Object Model (attribute, relationship, operation, exception, hierarchy, class and interface), whereas the latter integrates existing characteristics of schema versions into other schema versions also intervening on the DAG. All the supported statements correspond to operations that do not operate an update-in-place but always generate a new schema version. All these operations act on the current schema version selected via the `SET SCHEMA` statement (see Subsection3.1).

The first collection includes a complete set of operations [1] handled by the `CREATE`, `DROP` and `ALTER` commands modified to accommodate the extensions:

`CREATE` <element specification> [<schema change validity>]

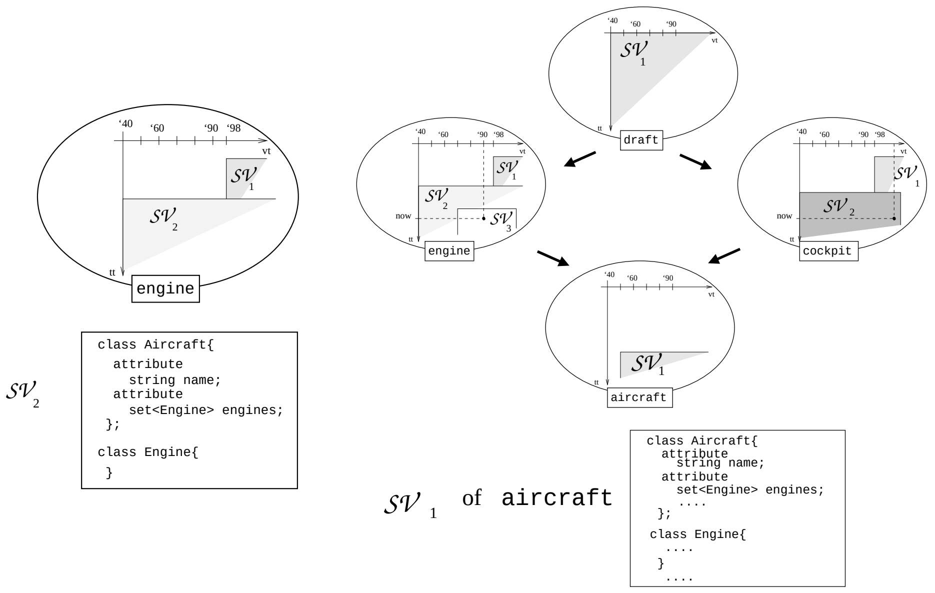
`DROP` <element name> [<schema change validity>]

`ALTER` <element to alter> [<schema change validity>]

The main extension concerns the possibility of specifying the validity of the new schema version (the full syntax of unexpanded non-terminals can be found in Appendix A). The outcome of the application of any of these schema changes is a new schema version with the same label of the affected schema version and the validity specified, if the <schema change validity> clause exists, $[-\infty, \infty]$ otherwise. Suppose that, in the engineering company example, the team working on the "engine" part is interested in adding a new class called "Engine" and an attribute in the class "Aircraft" to reference the new class. This can be done by means of the following statements (the outcome is shown on the left side of Figure 2):

```
SET SCHEMA LABEL engine
   AND VALID AT DATE '1999-01-01';
CREATE CLASS Engine VALID PERIOD '[1940-01-01 - forever]';
CREATE ATTRIBUTE set<Engine> engines IN Aircraft
VALID PERIOD '[1940-01-01 - forever]';
```

The second collection includes the following statements (the full syntax can be found in Appendix A):



```
class Aircraft{
  attribute
    string name;
  attribute
    set<Engine> engines;
 };

class Engine{
 }
```

```
class Aircraft{
  attribute
    string name;
  attribute
    set<Engine> engines;
    ....
 };
class Engine{
  ....
 }
  ....
```

Fig. 2. Outcomes of the application of modification statements

`ADD` <element to add> `FROM SCHEMA` <schema selection condition>
[<schema change validity>]

`MERGE FROM SCHEMA` <schema selection condition>
[<schema change validity>]

The `ADD` statements can be used for the integration of populated elements, like attributes or relationships, in the affected schema version, whereas the `MERGE` statement can be used for the merging of two entire schema versions. They originate from the CAD/CAM field where they are necessary for a user-driven design version control. They implicitly operate on the DAG by binding the involved nodes by means of edges. Both statements require two schema versions: the affected schema version, selected via the `SET SCHEMA` statement, and the source schema version from which the required characteristics are extracted or are to integrate, selected via the <schema selection condition> clause. Notice that the main difference between the use of the `ADD` and the `CREATE` statements is that the former consider populated elements (with values inherited) belonging to the source schema version, while the latter simply adds new elements with a default value. In our airplane design example, two consolidated schema versions from the nodes "engine" and "cockpit" can be merged to give birth to a new node called "aircraft". This last represents the final state of the design process:

```
CREATE SCHEMA aircraft
   FROM SCHEMA LABEL engine AND VALID AT DATE '1990/01/01'
VALID PERIOD '[1950-01-01 - forever]';
```

```
SET SCHEMA LABEL aircraft
   AND VALID AT DATE '1999-01-01';
MERGE FROM SCHEMA LABEL cockpit
   AND VALID IN PERIOD '[2010]'
VALID PERIOD '[1950/01/01 - forever]';
```

The outcome is shown on the right side of Figure 2. The first statement creates a new node called "aircraft" by making a copy of the current schema version belonging to the "engine" node valid at '1990/01/01'. Then, the `SET SCHEMA` statement sets the default values for label and valid time. The last statement integrates in the selected schema version the schema version belonging to the "cockpit" node valid in 2010. The temporal pertinence to be assigned to the new schema version is $[1950/01/01, \infty]$.

3.4 Data manipulation operations

Data manipulation statements (retrieval and modification operations) can use different schema versions if preceded by appropriate `SET SCHEMA` instructions. However, we propose that single statements can also operate on different nodes at the same time. To this purpose, labels can also be used as prefixes of path expressions. When one path expression starts with a label of a node, such a node is used as a context for the evaluation of the rest of the path expression. For instance, the following statement:

```
SELECT d.name
   FROM draft.Aircraft d, cockpit.Aircraft c
      WHERE d.name=c.name
```

retrieves all the aircraft names defined in the initial design version labelled "draft", which are still included in the successive "cockpit" design version. The expression `draft.Aircraft` denotes the aircraft class in the node labelled "draft", while the expression `cockpit.Aircraft` denotes a class with the same name in the node "cockpit". When the label specifier is omitted in a path expression, the default label set by the latest `SET SCHEMA` (or the global default) is assumed.

3.5 Transactions and schema changes

Transactions involving schema changes may also involve data manipulation operations (e.g. to overcome default mechanisms for propagating changes to objects). Therefore, schema modification and data manipulation statements can be interleaved according to user needs between the `BEGIN TRANSACTION` statement and the `COMMIT` or `ROLLBACK` statements. The main issues concerning the interaction between transactions and schema modifications are:

– the semantics of *now*, that is, which are the current schema versions which can undergo changes and which is the transaction-time pertinence of the new schema versions of the committed transactions,

– how schema modifications are handled inside transactions, in particular on which state the statements which follow a schema modification operate.

A typical transaction has the following form:

```
BEGIN TRANSACTION
SET SCHEMA LABEL ℓ
   AND VALID ss_v
      AND TRANSACTION ss_t;
...
COMMIT WORK
```

We assume that the transaction-time tt_T assigned to a transaction T is the beginning time of the transaction itself. Due to the atomicity property, the whole transaction, if committed, can be considered as instantly executed at tt_T. Therefore, also the *now* value can always be considered equal to tt_T while the transaction is executing. Since "now" in transaction time and "now" in valid time coincide, the same time value tt_T can also be used as the current (constant) value of "now" with respect to valid time during the transaction processing. In this way, any statement referring "now"-relative valid times within a transaction can be processed consistently. Notice that, if we had chosen as tt_T the commit time of T, as other authors propose [7], the value of "now" would have been unknown for the whole transaction duration and, thus, how to process statements referencing "now" in valid time during the transaction would have been a serious problem. Moreover, since $now = tt_T$ is the beginning of transaction time, the new schema versions which are outcomes of any schema modification are associated with a temporal pertinence which starts at tt_T.

When schema and data modification operations are interleaved in a transaction, data manipulation operations have to operate on intermediate database states generated by the transaction execution. These states may contain schemas which are incrementally modified by the DDL statements and which may also be inconsistent. A correct and consistent outcome is, thus, the developer's responsibility. The global consistency of the final state reached by the transaction will be automatically checked by the system at commit time: in case consistency violations are detected, the transaction will be aborted.

4 Conclusions and Future work

We have proposed extensions to the ODMG data definition and manipulation languages for generalized schema versioning support. This completes the ODMG extension initiated in [4–6] with the work on the data model. The proposed extensions are compatible with SQL-92 and TSQL2, as much as possible, and include all the primitive schema change operations considered for the model and, in general, in OODB literature. Future work will consider the formalization of the semantics of the proposed language extensions, based on the algebraic operations proposed in [6]. Further extensions could also consider the addition of other schema versioning dimensions of potential interest, like spatial ones [9].

References

1. J. Banerjee, W. Kim, H.-J. Kim, and H. F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *Proc. of the ACM-SIGMOD Annual Conference*, pages 311–322, San Francisco, CA, May 1987.
2. R. G. G. Cattell, D. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, H. Strickland, and D. Ware, editors. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann, San Francisco, CA, 1997.
3. C. De Castro, F. Grandi, and M. R. Scalas. Schema Versioning for Multitemporal Relational Databases. *Information Systems*, 22(5):249–290, 1997.
4. F. Grandi, F. Mandreoli, and M. R. Scalas. A Formal Model for Temporal Schema Versioning in Object-Oriented Databases. Technical Report CSITE-014-98, CSITE - CNR, November 1998. Available on ftp://csite60.deis.unibo.it/pub/report.
5. F. Grandi, F. Mandreoli, and M. R. Scalas. Generalized Temporal Schema Versioning for GIS. Submitted for publication, 1999.
6. F. Grandi, F. Mandreoli, and M. R. Scalas. Supporting design and temporal versions: a new model for schema versioning in object-oriented databases. Submitted for publication, 1999.
7. C. S. Jensen, J. Clifford, S. K. Gadia, P. Hayes, and S. Jajodia et al. The Consensus Glossary of Temporal Database Concepts - February 1998 Version. In O. Etzion, S. Jajodia, and S. Sripada, editors, *Temporal Databases - Research and Practice*, pages 367–405. Springer-Verlag, 1998. LNCS No. 1399.
8. C. S. Jensen, M. D. Soo, and R. Snodgrass. Unifying Temporal Data Models via a Conceptual Model. *Information Systems*, 19(7):513–547, 1994.
9. J. F. Roddick, F. Grandi, F. Mandreoli, and M. R. Scalas. Towards a Model for Spatio-Temporal Schema Selection. In *Proc. of the DEXA'99 STDML Workshop*, Florence, Italy, August 1999.
10. J. F. Roddick and R. T. Snodgrass. Schema Versioning. In *[11]*.
11. R. T. Snodgrass, editor. *The TSQL2 Temporal Query Language.* Kluwer Academic Publishers, 1995.

A Syntax details

The non-terminal elements which are not expanded in this Appendix can be found in the BNF specification of ODMG ODL [2] and TSQL2 [11].

```
<element specification>::=
        <element specification in interface> IN <interface name>
   |    <interface specification>

<interface specification>::=
        INTERFACE <interface name>
   |    CLASS <class name>

<element specification in interface> ::=
        ATTRIBUTE <domain type> <attribute name> [<fixed array size>]
   |    RELATIONSHIP <target of path> <relationship name>
          INVERSE <interface name>::<relationship name>
```

```
    |      OPERATION <op type spec> <operation name> <parameter dcls>
             [RAISES(<scoped name list>)] CODE <code spec>
    |      EXCEPTION <exception name> TO OPERATION <operation name>
    |      EXCEPTION <exception name> {[<member list>]}
    |      SUPERINTERFACE <interface name>

<element name>::=
           <element name in interface> IN <interface name>
    |      <interface specification>

<element name in interface> ::=
           ATTRIBUTE <attribute name>
    |      RELATIONSHIP <relationship name>
    |      OPERATION <operation name>
    |      EXCEPTION <exception name> TO OPERATION <operation name>
    |      EXCEPTION <exception name>
    |      SUPERINTERFACE <interface name>

<element to alter>::=
           <element to alter in interface> IN <interface name>
    |      INTERFACE NAME <interface name> INTO <interface name>
    |      CLASS NAME <class name> INTO <class name>

<element to alter in interface>::=
           ATTRIBUTE NAME <attribute name> INTO <attribute name>
    |      ATTRIBUTE TYPE <attribute name> INTO <domain type>
    |      RELATIONSHIP NAME <relationship name> INTO <relationship name>
    |      RELATIONSHIP TYPE <relationship name> INTO <target of path>
    |      INVERSE TYPE <relationship name> INTO <relationship name>
    |      OPERATION NAME <operation name> INTO <operation name>
    |      OPERATION CODE <operation name> INTO INPUT <op type spec>
             OUTPUT  <parameter dcls> CODE  <code spec>
    |      EXCEPTION NAME <exception name> INTO <exception name>
    |      EXCEPTION TYPE <exception name> INTO {[<member_list>]}

<element to add>::=
           <element to add in interface> IN <interface name>
    |      <interface specification>

<element to add in interface> ::=
           ATTRIBUTE <attribute name>
    |      RELATIONSHIP <relationship name>
    |      OPERATION <operation name>
    |      EXCEPTION <exception name>
```