STRUCTURE OF A PRODUCT DATABASE SUPPORTING MODEL EVOLUTION

by C. M. Eastman, H. Assal, T. Jeng[1]

ABSTRACT: One of the primary capabilities for integrated data models and databases for design is support for dynamic evolution. The need is to support design changes that affect the schema, and often existing objects already loaded with data. We present scenarios using EDM-2, a database providing these capabilities.

1. INTRODUCTION:

A growing consensus is that the most desirable way to support computer integration in architecture and the construction industry is to rely on a framework supporting a modular, extensible building model [Augenbroe and Rombouts,1994], [Galle,1994]. Application modules represent different knowledge areas: different construction technologies, building types, special building functions or performances, construction planning, or special space uses. The various application modules are assembled and integrated into a building model as needed -- at the beginning of a project, and/or dynamically over the life cycle of the building. Such an approach addresses the combinatorial issues regarding the rich range of technologies and building types that comprise the building industry. It supports innovation and the application of unique combinations of technologies and knowledge in particular building projects. It also encourages development and use of applications applying new construction methods, new types of space use, new kinds of building code issues [Eastman, 1993]. The need for extensibility is also being explored in the ISO-STEP standards committees, in recognition of its importance in supporting lifetime product models.

Other work has addressed schema evolution of databases. Banerjee and Kim et al, [1987] laid out schema operations for adding to and deleting classes and attributes within a schema when instances are loaded. Batini et al [1986] reviewed methods for merging multiple schemas, but without data. All methods to date provide means for making low level extensions, not how to add or merge modules on a database with loaded data. They provide tools for the system developer. Our goal is to make schema extension available to users.

---

[1] Center for Design and Computation and Department of Architecture, University of California, Los Angeles, CA. 90024-1467. The principal author can be contacted through email at *chuck@gsaup.ucla.edu.*

Construction Informatics Digital Library http://itc.scix.net/
paper w78-1995-327.content

EDM-2 is a product database grounded on a data model supporting model evolution. Its design and implementation supports evolution in terms of adding new attributes to existing objects, interfacing new analyses or applications on an existing, fully loaded model, model restructuring involving moving objects from one class to another, and selective deletions and clean-up, as knowledge domains cease to be relevant.

This paper presents the schema and model evolution capabilities of EDM-2. It first gives an overview of the EDM-2 database system. It then describes the features for model evolution, and how extensions are made to both the model schema and object instances. It also presents the operations for adding modules or applications to a loaded model. Throughout, it reviews the features supporting model reduction. An example is used, of a concrete structure using post and beam and slab construction. The example is built up incrementally, showing the features of the EDM-2 database language. Corresponding to the EDM-2 database language is a graphical notation, called GML. GML is being implemented as a graphical front end for users. It will be used here to give an overview of the database structure, as it evolves throughout the example.

2. EDM-2

EDM-2 is a database management system based on a data model tailored for product model use [Eastman, Chase et al, 1993]. Its intended use is for the implementation of backend databases supporting the integration of a heterogeneous and evolving set of applications. It incorporates operations for data management and is not meant to support design operations; these are the responsibility of external applications. As a backend database, it addresses the following capabilities:

- translation of data between the database views corresponding to different application interfaces, in both read and write modes [Assal and Eastman, 1995];
- managing the integrity of data, especially among concurrent users making iterated decisions [Eastman, Cho et al, 1995];
- version control and iterating to earlier design stages; and
- model evolution, in support of new applications, as needed both during design and over the building life cycle.

This paper focuses on the functionality needed to support the last capability. Only a quick overview of EDM-2 is possible here. Technical descriptions of EDM-2 are presented in [EDM Team, 1995], [Eastman, Cho et al, 1995].

The EDM-2 data model is object-based; it represents objects that are defined using multiple inheritance of other object classes. In general, EDM-2 incorporates a class-instance structure. Attributes of an object also have the behavior of objects. All object classes have a global definition and if that definition is modified, all objects using that definition are also modified. Object classes defined with a single value are

called *Domains*, complex object classes are called *Design Entities* (*DE*s). Attributes may be constructed as variable length SEQUENCEs or SETs.

An important part of extensibility is the treatment of declarations and deletions. Ideally, once a complex object class has been defined, new object instances should have all the necessary sub-objects and attributes created automatically. Similarly, a deletion of a class or instance should delete all dependent objects automatically. Shared objects and complex data structures are a particular problem, where multiple references may abound. EDM-2 incorporates distinct structures for building up relations between multiple independent objects. It also has strong rules for managing instance creation and deletions. These structures and rules will be defined for the various classes of structure, as they are introduced.

2.1 Design Entities (DEs)

The general object class in EDM-2 is called Design Entity (DE). DEs may be used to define complex attributes (shape) and also physical objects or systems at various levels of abstraction (beam, window, an auditorium space). DEs have attributes, both simple and complex (defined by Domains and other DEs) and include Constraint calls and Mapcalls (to be defined shortly). They may be declared inheriting other DEs. Each DE class may be optionally defined to have a *Keyname*.

All DEs that are referenced as attributes of other DEs become dependent upon the referencing DE and only one such reference is allowed. To deal with DEs that are meant to be shared, they are declared as SHARED. When a DE instance or class is deleted, all non-shared attributes of the DE are also deleted.

2.2 Composition

The aggregation of heterogeneous DE classes to define another DE class is represented by the *Composition*. The aggregated DE of a Composition is called the *target* and the component DEs are called *parts*. There may be multiple Compositions describing the same target, for example one set of finite elements to define the thermal performance of a shape and another set to define the structural performance. The same DE may be a part in multiple Compositions. For example a pump may be part of the fluid flow Composition and also the electrical Composition. Compositions may be defined top-down or bottom-up, that is parts-first or target-first. Compositions have Ccalls associated with them, defining rules of analysis or design rules for the Composition.

Compositions, like DEs, have associated constraint calls and Mapcalls. These define rules of composition, rules required for the application of some analysis or other criteria associated with the composition.

Compositions are considered a special property of their target DE. As a result, when a DE class that is the target of a Composition has instances created, each instance has an associated Composition instance. Also, if a DE, that is the target of a Composition, is specialized or declared as an attribute into another DE, the Composition is automatically specialized also. The reverse is not true; when a DE that is the target of a composition is deleted, the Composition remains, allowing it to stand alone or be associated with another target DE. The part DEs are not acted upon in specialization; a newly specialized composition is initialized without part DE instances. Composition structures may also be specialized. Thus a Composition defining a planar polyhedron solid model would have planar polyhedra as parts. This Composition can be specialized to include NURBS faces, in addition to planar ones. Composition specialization allows the adding of new Parts, Ccalls and Mapcalls.

2.3 Constraints

Integrity is the relation between some aspect of reality and any model of it. If the model corresponds to reality, we say that it has integrity [Ullman, 1988]. Integrity is generally defined by a set of correspondence rules. EDM-2 uses constraints to represent those rules. Constraints are relations defined in an application interface, or by a user to determine if a relation is satisfied or not.

Constraints are defined in two parts: the *Constraint* is the rule specification; the constraint call or *Ccall* is the specific invocation site. The Constraint specifies the argument types and when it exists, the constraint body, written as a C function that can be dynamically linked into the database. The Ccall provides the specific arguments at a call site. Ccalls have a truth value and are of two types. One type is *invariant*, which are evaluated automatically and must have a value of True (or Undefined, if needed data is missing) at the end of a transaction. They are used to deal with low level issues of the model structure and to maintain referential integrity. They are checked automatically at the end of any transaction, before a COMMIT is completed. If any invariant Ccall fails, the transaction is rejected. The other constraint type is *variant*, which may have a value of True, False or Unknown and is evaluated under user command. Variant constraints are used to define a product model's integrity in terms of rules required for application to execute, physical laws, well-formedness conditions, such as non-overlap, or building codes and trade practices. Variant Ccalls are assumed to possibly be false for extended periods during design. and are explicitly evaluated by users.

2.4 Maps

Another feature of EDM-2 are objects that derive data from other data [Assal and Eastman, 1995], called *Maps*. Maps are related to Constraints and can be used in several ways:

- to translate data in one representation to another representation
- to derive dependent data
- to propagate constraint relations from one set of data to another.

Maps are defined in a similar manner to Constraints, they have a specification, called a Map, and a call site, defined by a Mapcall. Mapcalls also have an integrity value, telling if the data generated is current. Maps are called on demand, similar to variant Ccalls.

2.5 Accumulation

The Accumulation structure defines a relation over a set of Ccalls, within either a DE or Composition. It defines two sets of Ccalls: a *pre-condition set* and a *post-condition set*. The pre-condition set must be satisfied before the post-condition set can be meaningfully evaluated. Thus an accumulation defines a precedence ordering over Ccalls.

The class definition of an Accumulation can represent the interface and semantics of complex rules and operations. It provides one part of an interface for external applications. It identifies the well-formedness constraints (pre-conditions) that must be satisfied in order for the application to properly execute. The Application itself corresponds to the post-condition constraints, which defines the relations the application satisfies. The use of Accumulations will be made clearer in the example.

3. The Example

Our example is of a subsystem of a building, a simple structural system using concrete slabs, beams, girders and columns. In this limited space of this paper, only a schematic portion of the system can be delineated.

In order to characterize the need and uses of evolution in a building model, we outline here a simple design development scenario. The scenario unfolds incrementally. The building model can be shown diagrammatically as in Figure One. The structural system is defined first at the abstract level, then geometrically as a grid (which may be used by other systems also), using a grid layout application. With the grid as reference, parts of the structure are defined, here just slabs. A slab application computes the layout and thickness of the slabs. Later, because of site limitations, the design is required to accept a skewed gridline. This requires both a schema change on data that has already been used in deriving other data. It also requires value changes at the slab definition level. Other model changes affect the attribute definitions of already created instances. Initially the design was based on an external stairway, outside of any grid panels. But the client changes their mind and move the stair inside

one of the grid cells. This requires another schema extension to support slabs with holes and moving an existing slab instance to this new class.

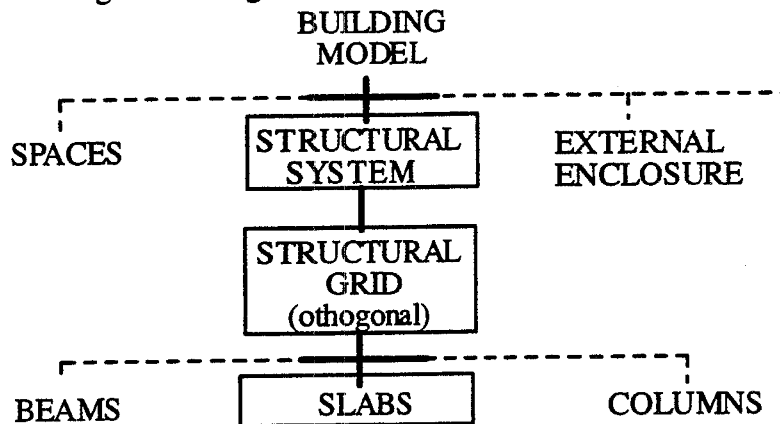

FIGURE ONE: The building model developed in the example.

Such a scenario is simplistic. However, most applications have limitations and special conditions often arise, similar to these, that must be addressed in an ad hoc fashion, requiring different applications, hand adjustments, and changes to the model schema. It is these capabilities that the example demonstrates.

3.1 Model Definition

The initial schema with is shown in GML in Figure Two. In GML, DEs are defined as enclosed boxes, with their definition shown inside. Classes are shown in bold boxes, instances in thin. Attributes are in italics, entities with an M prefix are Mapcalls and those with a C are Ccalls. K denotes a *Keyname* and S a *shared* DE. Keynames replace class names for instances. Inheritance is shown with light lines, Compositions by heavy, with an arc spanning parts. DE attribute references are shown with arrows. Compositions are named, with Accumulations and Ccalls below.

The corresponding code is shown in Table One. Domain types are defined first. The grid lines are defined initially as a general grid line, that is specialized into vertical and horizontal grids. The vertical and horizontal grid lines are grouped with a composition called `to_grid`, and aggregated into a DE called `grid`. The application `grid_applic` guarantees that the grid lines are in ascending order, and this assumption is used in later applications. An initial layout is defined, consisting of three vertical and three horizontal grid lines. The lines are first created, then inserted into a grid instance `grid1`.

A second application defines the slab DEs in terms of their geometry and thickness, defined by referencing the gridlines that bound them. The slabs are also grouped into

the `struct` Composition. The application defines three slab instances that are bordered by the gridlines. The application checks that the slabs are not overlapping. The application is characterized by an Accumulation, defining its pre-conditions and the conditions the application guarantees. The instance model is kept small here for illustrative purposes. The DML-2 graphical representation of this schema and the initial instances generated is shown in Figure Two, the EDM-2 code in Table Two. The intention for this model is that later defined slabs, beams and columns will be defined relative to the grid. If it is necessary to change a gridline's value, then all parts that refer to the changed line will be automatically updated.

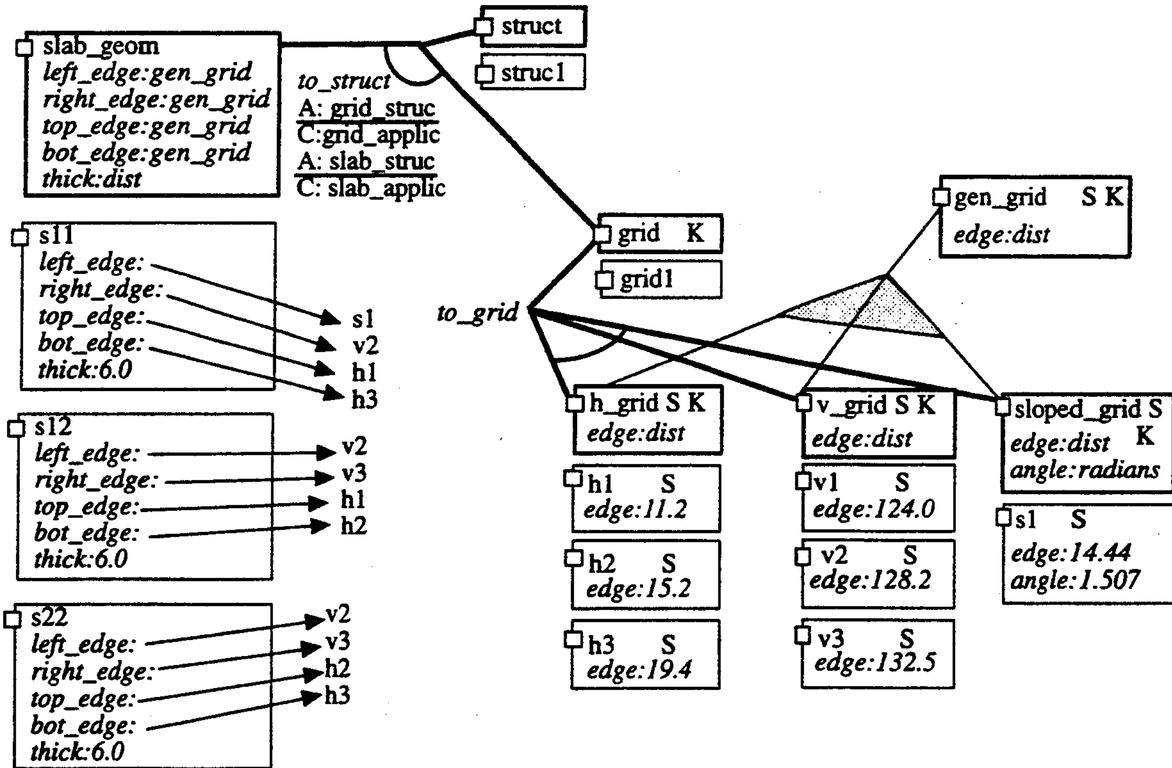

FIGURE TWO: The EDM-2 schema and instances after running the first application, characterized in GML format.

3.2. Model Evolution

Model evolution requires that new DEs and Domains classes can be created at any time, or existing ones can be added to or deleted from existing definitions, or structures completely eliminated. Also, instances that have references to them also need to be correctly deleted. Thus for each EDM structure: Domain, DE, Constraint and Ccall, Map and Mapcall, Composition and Accumulation, operators exist to create, add to, or totally remove the class definition. Since these operations can be

executed at any time, they not only affect changes to the class definition of entities, but also make equivalent changes to all affected instances, and to all specializations and attributes of DEs using the changed entity.

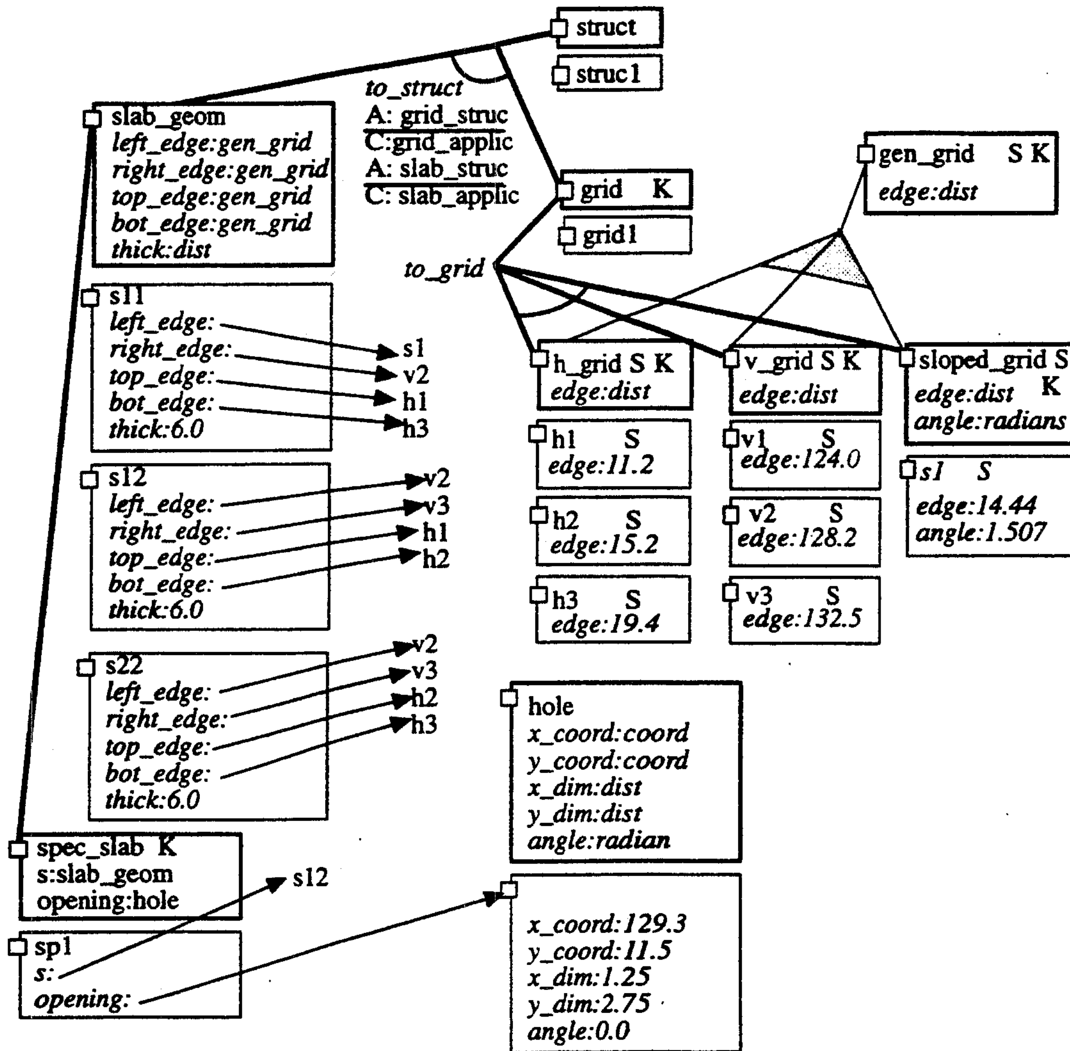

FIGURE THREE: The extended schema and instances after execution of the slab layout program.

Figure Three presents a small example of model evolution. The EDM-2 code is shown in Table Three. The original grid is extended to accept skewed grid lines. These are defined geometrically with an x-coordinate intercept and angle. A `sloped_grid` class is defined as a specialization of `gen_grid` and is added to the composition An instance is created. One slab is modified to be bordered by the

sloped gridline. Since the type of the border is `gen_grid`, the sloped gridline can be assigned it.

3.4 Instance Modifications

It is also often required that a building element take on a new function, or require a more elaborate geometric or property specification. In this case, a new type is required and instances that are already existing need to be moved to it. In EDM-2, DE instances may be migrated up or down a specialization lattice, with the effect that new attributes may be added to or deleted from existing DE instances. This allows attributes to be added or deleted well into the lifetime of an object instance, without losing needed references to the instance.

In our example, this is required when it is decided that one of the instantiated slabs must have a hole added to it, defined within a single slab. The opening is here motivated by the addition of an external stairway. The opening is defined as a specialization of `slab_geom`, using `slab_geom` as a nested attribute. The one slab instance is migrated (specialized) to this new class and the data fields assigned to represent the opening. The EDM-2 code describing these changes are shown in Table Four and the GML diagram is shown in Figure Three. It should be noted that all references to the changed instances remain intact.

4. APPLICATION INTERFACES

It is assumed that all external applications will have an associated application view, that either corresponds one-for-one, isomorphically, with the internal data structures of the application, or is possibly a subset of the internal structures. These views are inserted into a model as the corresponding application is found to be needed. In the example, the original grid schema corresponds to the view of the initial grid application; the slab defining application was added second. For ease of the example, it was assumed that only subsets of the application were used initially and then they were expanded as needed. Alternatively, the initial view could have been mapped into a completely different representation to support more powerful capabilities.

In general an application view includes the following information:

- the DE objects, needed to represent the objects;
- Maps and Mapcalls for extracting data from the model into the view and later for translating the data back [Assal and Eastman, 1995];
- a structure that defines the needed integrity for the application, in terms of conditions in the data that must be satisfied for the application to properly execute; here we use an Accumulation;
- constraints that represent these integrity conditions.

5. CONCLUSION

We have outlined the language capabilities supporting model evolution implemented in EDM-2. We expect to expand the implementation to test it in production-style applications. We believe that the data model and its associated implementation techniques can serve as a model for other data models and databases seeking to achieve lifetime model evolution.

NOTE: This work was supported by NSF grant, No. IRI-9319982.

REFERENCES

Augenbroe,G and W. Rombouts. [1994] "Extended Topology in Building Design Systems" ASCE 1st Congress on Computing.,ASCE, Washington, D.C.

Assal, H and C. Eastman, [1995] "Engineering Database as a Medium for translation", 1995 CIB W-78 Symposium, Stanford, Calif.

Banerjee, J. W. Kim, H. Kim and H. Korth [1987] "Semantics and implementation of schema evolution in object-oriented databases",Proc. ACM SIGMOD 1987 Annual Conf. 16:3, pp.311-322.

Batini, C., M. Lenzerini et al,[1986], "A comparative analysis of methodologies for database schema integration" ACM Comp. Surv., 18:4, (Dec.), pp.323-364.

Eastman, C.M., S. Chase and H. Assal, [1993], "System architecture for computer integration of design and construction knowledge", Automation in Construction, 2:2, (July), pp. 95-108.

Eastman, C.M. [1995] "Managing Integrity in Design Information Flows" Design and Computation working paper, G.S.A.U.P., University of California, Los Angeles, CA. 90024.

Eastman, C.M., Cho, M.S., Jeng, T.S., Assal, H., [1995] "A Data Model And Database Supporting Integrity Management", ASCE Intern. Computing Congress, Atlanta, GA.

EDM Team, [1994] EDM-2 Reference Manual, Research Report, Center for Design and Computation., University of California, Los Angeles, CA. 90024

Galle, Per, [1994] "Towards integrated 'intelligent' and compliant computer modeling of buildings", Report of Computer Aided Building Design Unit, Technical University of Denmark, Lyngby, Denmark.

Ullman, Jeffrey D. [1988], Principles of Database and Knowledge-Base Systems, Vol.1, Computer Science Press, Rockville, MD.

```
/* SCHEMA EXTENSION */
/*layout of construction grid */
CREATE DOMAIN dist:REAL
  LBOUND 0.0
  DESC "distance in meters";
CREATE DOMAIN coord:REAL
  DESC coordinates in meters;
CREATE DE struct KEYNAME
DESC "DE to define structure";
CREATE COMP to_struct
  TARGET struct
  DESC "comp. to describe structure";
/* create structure instance */
:st1=INSERT INTO DE struct
  KEYNAME=struc1;
/* create a general gridline and two
  specializations of it */
CREATE DE gen_grid SHARED KEYNAME
    ATTR (edge:dist);
CREATE DE v_grid
  INHERIT(gen_grid);
CREATE DE h_grid
  INHERIT(gen_grid);
CREATE DE grid KEYNAME
  DESC "the aggregate structure for a
  complete grid";
/*add grid as component of structure*/
MODIFY COMP struct
ADD PART(grid);
CREATE COMP to_grid
  TARGET grid
  PART(SEQUENCE OF v_grid,
        SEQUENCE OF h_grid)
  DESC "composition to group gridlines";
CREATE CONSTRAINT g_a(
        SEQUENCE OF v_grid,
        SEQUENCE OF h_grid)
DESC "constraint and ccall to shadow
application puts in monotonic order";
CREATE CCALL grid_applic
  CONSTRAINT g_a
  (part_grid.comp_to_grid.part_v_grid,
 part_grid.comp_to_grid.part_h_grid);
CREATE ACCUM grid_struc
  REF to_struct
  POST (grid_applic)
  DESC "accum.is shadow for grid applic";
/*APPLICATION RUN: an example layout */
:v1=INSERT INTO DE v_grid KEYNAME=v1
  (edge=124.0);
:v2=INSERT INTO DE v_grid KEYNAME=v2
  (edge=128.2);
:v3=INSERT INTO DE v_grid KEYNAME=v3
  (edge=132.5);
:h1=INSERT INTO DE h_grid KEYNAME=h1
  (edge=11.2);
:h2=INSERT INTO DE h_grid KEYNAME=h2
  (edge=15.2);
:h3=INSERT INTO DE h_grid KEYNAME=h3
  (edge=19.4);
/* create grid instance - composition
to_grid made automatically */
:g1=INSERT INTO DE grid KEYNAME=grid1;
/* add parts to to_grid composition */
UPDATE TARGET=:g1 OF COMP to_grid
  ADD PART (v_grid=KEYNAME=(v1,v2,v3),
  h_grid=KEYNAME=(h1,h2,h3));
/*add parts to to_struc composition*/
UPDATE KEYNAME=struc1 OF COMP to_struc
ADD PART(grid=KEYNAME=(g1));
```

TABLE ONE: Initial setup of grid and populating it. The grid layout application is shadowed by the accumulation `grid_struc`.

```
/* SCHEMA EXTENSION    */
CREATE DE slab_geom KEYNAME
  ATTR (left_edge:gen_grid,
  right_edge:gen_grid,
  top_edge:gen_grid,
  bot_edge:gen_grid,
  thick:dist )
  DESC "slab corresponds to multiple
  cells in grid, with thickness";
CREATE CONSTRAINT slab_a
        (SEQUENCE OF v_grid,
        SEQUENCE OF h_grid,slab_geom)
DESC "constraint & ccall shadows applic";
CREATE CCALL slab_applic
  CONSTRAINT slab_a
  (part_grid.comp_to_grid.part_v_grid,
   part_grid.comp_to_grid.part_h_grid,
                        part_slab_geom);
CREATE ACCUM slab_struc
  REF to_struct
  PRE (grid_applic)
  POST(slab_applic)
DESC accumulation to define precedence
relations;
/*APPLICATION RUN:  example instance slab
definitions */
:s11=INSERT INTO DE slab_geom
  KEYNAME=s11
  (left_edge= :v1,
  right_edge= :v2,
  bot_edge= :h1,
  top_edge= :h3 );
:s12=INSERT INTO DE slab_geom
  KEYNAME=s12
  (left_edge= :v2,
  right_edge= :v3,
  bot_edge= :h1,
  top_edge= :h2 );
:s22=INSERT INTO DE slab_geom
  KEYNAME=s22
  (left_edge= :v2,
  right_edge= :v3,
  bot_edge= :h2,
  top_edge= :h3 );
/* update to_struct composition */
UPDATE COMP to_struc
  ADD PART(SEQUENCE OF slab_geom);
/* make composition instance for
application run*/
UPDATE TARGET=KEYNAME=str OF COMP
to_struct
ADD PART(slab_geom=KEYNAME(s11,s12,s22));
```

TABLE TWO: EDM-2 code of the schema and interface for an application that lays out floor slabs.

```
/* SCHEMA EXTENSION        */
CREATE DOMAIN radians:real
DESC "angle in radians";
/*sloped grid is spec. of gen_grid*/
CREATE DE sloped_grid
  INHERIT:gen_grid,
  angle:radians )
  DESC "a sloped grid line defined as an
  angle and the x intercept";

/* add the sloped gridline to grid
  comp. and */
MODIFY COMP to_grid ADD PART(sloped_grid);

/*update slab_geom to accept sloped
gridlines */
MODIFY DE geom_slab ADD ATTR
(sl_g:sloped_grid);

/* create sloped grid instance */
:sl=INSERT  INTO DE sloped_grid KEYNAME=sl
   (angle=1.5707, edge=14.44);

/* add sloped instance to grid comp */
UPDATE TARGET=KEYNAME=grid1 OF COMP
to_grid
   ADD PART (sloped_grid = (:sl));
/* modify slab instance to refer to skewed
gridline */
UPDATE :sl1 OF DE slab_geom
(left_edge = :sl);
```

TABLE THREE: Schema modification, adding skewed gridline capability to the previously defined and populated grid.

```
/* SCHEMA EXTENSION     */
CREATE DE hole
ATTR (x_coord:coord,
       y_coord:coord,
       x_dim:dist,
       y_dim:dist,
       angle:radians)
DESC "slab opening" ;

/* specialize slab to hold opening */
CREATE DE spec_slab KEYNAME
ATTR (s:slab_geom,
      opening:SEQUENCE OF hole);
/* put special slab in comp struct */
MODIFY COMP struct ADD PART (spec_slab);

/* APPLICATION RESULTS   */
/* move existing slab instance to de that
can include holes  */
SPECIALIZE (KEYNAME=(sl2)) OF DE slab_geom
to spec_slab (KEYNAME=sp1);

UPDATE KEYNAME=sp1 OF DE spec_slab
(opening.x_coord=129.3,
opening.y_coord=11.5,
opening.x_dim=1.25,
opening.y_dim=2.75;
```

TABLE FOUR: An extension specializing an existing slab instance to include one or more openings. An instance of a slab is then moved to this new DE class.