

## **Chapter 9**

### **Schema Instance Management**

In an integrated design system instances of evolving schemas must be able to be quickly and correctly created and maintained. This aids in the testing and validation of both the schemas and the inter-schema mappings, and helps to reduce schema development time. Tools are needed to instantiate sample building models for evolving schemas, and to navigate the resultant structures to ascertain that they meet their required purpose. To a large extent this requires tools which mimic the functionality of the tools (DTs) that will be part of the final integrated system. This leads to a conflict between the need for tools which can quickly create and maintain complete models for the developing schemas, versus avoidance of effort spent developing tools specialised for a particular schema, which is likely to change, and will be discarded at the end of the development phase. Generic tools are required which are either independent of the developing schemas, or quickly tailorable for specific schemas where independence is not achievable. A set of requirements for this type of tool is detailed below. Four tools of this type which have been developed or used in this project are described in this Chapter.

#### **9.1 Requirements for Instance Management**

To enable instance creation and maintenance in the development environment the tools used must satisfy a diverse range of requirements. The main requirements are covered in Section 8.2.1, two smaller requirements are described below:

**Genericity:** instances for a wide range of schemas need to be created, maintained and navigated.

Tools must be independent of the developing schemas, or easily adaptable. Developers will not wish to spend large amounts of time customising tools for the different schemas, so tools which adapt to specified schema definitions will be of great benefit.

Persistence: as the schema instance test set may become very large (e.g., for a complete building), it is preferable that instance maintenance and navigation tools maintain a persistent representation of the data independently of the ASCII data transfer format (ISO/TC184 1994) used to communicate between design tools in the final system. This will save a significant amount of loading, validation, and saving time when working with instances of a model.

## 9.2 Instance Management Systems

To work with the modelling environments described in Section 2.2, the schema instance management system has to comprehend EXPRESS schema definitions and data models sent in the associated STEP Part 21 data-file format (ISO/TC184 1994). The tools built for the instance management system in this project all operate in environments which provide parsers for these formats, and allow persistent databases to be created and manipulated according to the resultant specifications and data. These instance management tools meet the majority of requirements in Section 9.1, including instance manipulation and modification in a persistent store, viewing and navigation in both textual and graphical formats, and guaranteed consistency of the persistent store for data being manipulated in multiple views. The four tools created or modified for use in this project for instance creation and navigation are described below.

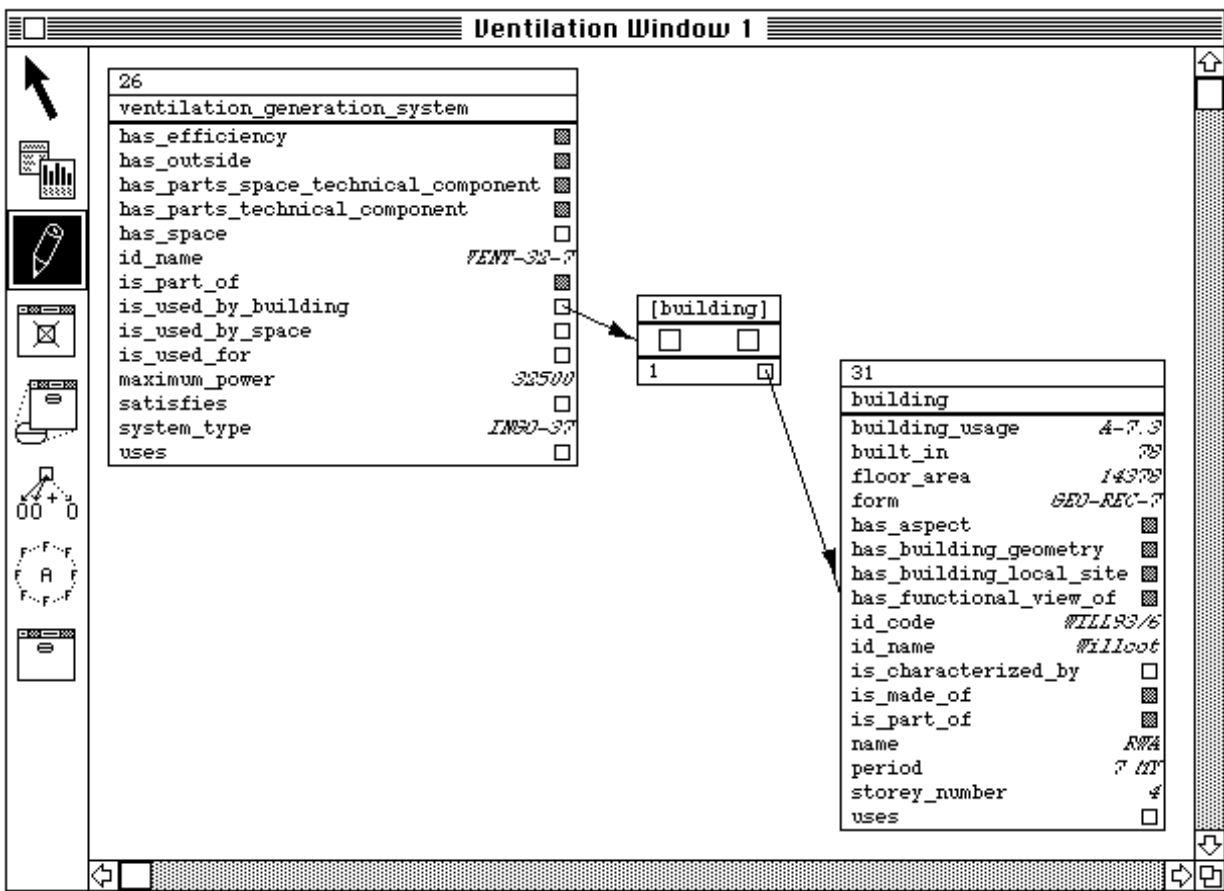
### 9.2.1 EPE: an instance construction and browsing system

The EPE system (Amor et al 1995; and as described in Section 3.2), developed to satisfy the requirements of Section 2.2.1, also supports implementation and maintenance. A schema developed in EPE can, at any time, be compiled to an underlying object-oriented form (Snart), and instances of the entities can be created and manipulated. Whole instance models, in the form of STEP data transfer format files, may be loaded into and transferred out of the system. Figure 9.1 shows an instance view of the *ventilation\_generation\_system* entity modelled in a previous COMBINE IDM.

Instance views consist of visual renditions of entity instances listing their attributes, each attribute's value (where one exists), and the links between entity instances. Navigation through the instance model is via the entity instance links, which are displayed as small boxes in an instance view. These links can be expanded to view the data contained in the linked instances. The amount of data seen for each entity instance can be controlled by the user to present views which show, for example, all instantiated attributes, or all attributes which are references, or all attributes with a basic type, or some combination of types. In Figure 9.1, all attributes of an instance of a *ventilation\_generation\_system* are shown, including links which are not instantiated (shown as greyed boxes). This figure also shows the single reference to a building contained in the *is\_used\_by\_building* attribute, and all the information in that instance of a building.

The user also has control over the way in which the information in entity views is laid out. This is defined through the use of a special purpose display language. This display language allows for displays which present sets of information in a variety of useful forms, such as bar graphs and tables, and allows templates of a given layout to be created.

In a similar fashion to the requirements for analysis and design views in the schema modelling section, multiple instance views with overlapping information can be created. The elements (e.g., entity attributes) in instance views are editable, allowing changes to be made to the underlying model instance. EPE's instance editor is a specialisation of CernoII, a run-time debugger and visualisation system developed as a companion for SPE (Fenwick et al. 1994; Grundy et al. 1993).



**Figure 9.1** Instance viewing and navigation

The original CernoII environment provided most of the functionality initially required for this project. However, extensions were made to specialise the system for the types of model and domain the system had to be tested in. The main extension was to allow instances of a model to be loaded, or saved, in the standard form used in model development in the building domain (i.e., STEP Part 21 data-files, ISO/TC184 1994). Another extension was to add the ability to view and modify facet information for any attribute of an object. This facet information includes the unit of an attribute's value, along with where the value was derived from, and constraints on its value. A direct manipulation tool was also incorporated which allows objects to be created and automatically linked to an attribute's value, or into a set, bag or list of object references for an attribute.



The textual browsing and editing of the model is performed on an object by object basis, though the user may follow relationships between objects in a hypertext like fashion. The graphical view displays a selected portion of the model, and the user can set the level of navigation and selection at which the tool will operate. In Figure 9.2 the user is navigating through the graphical model at a wall level, though other object types are displayed for reference.

In COMBINE this tool provides feedback to the conceptual task. Once a conceptual schema has reached a more or less stable version, DT teams and anyone who has a direct interest in the IDM are encouraged to browse instances of the schema and suggest modifications. In this thesis InSTEP was mainly used to ensure the validity of graphical representations in STEP data-files being developed for demonstration purposes.

### 9.2.3 SmartQuery and the ObjectViewer

The Smart language and development environment have been extended by the author to provide query and visualisation tools, as an adjunct to the environments described above. The query language allows a data model to be searched for objects matching a general query term. The ObjectViewer allows a dynamically updated view of an object to be created, and includes buttons for invoking object methods or allowing object modification.

SmartQuery allows a named data store to be searched, and for sets of objects matching several classes to be returned (i.e., response to a query is not limited to a single class type). The query can consist of attribute references, object ID references, referenced attributes (i.e., following pointer chains) and method calls which can be treated in a functional manner (i.e., either true or false or returning a single value). Compound query terms can be constructed with full arithmetic calculation, including Prolog predicates and aggregating functions (i.e., sum, count, minimum, maximum, average). List and array elements can also be accessed individually in the query. The returned list of object tuples matching the classes specified in the query can be utilised to invoke the ObjectViewer or the EPE view navigation environment. Invoking SmartQuery can be performed by accessing a menu item in the LPA Prolog environment or through an equivalent predicate call. Figure 9.3 shows both a query dialogue and the query result in the working window. A complete description of the query language can be found in Appendix C.2.

The ObjectViewer provides a general purpose debugging tool for the Smart language as well as a data model viewing and navigation tool. The basic premise of the ObjectViewer is that it spies upon an object, and always reflects the current state of the object. It is more powerful than that, as it also allows attributes to be modified, and methods of the class to be invoked, from the object view window. The ObjectViewer system allows a layout template for attributes and methods of a class to be specified when the class is defined, or by default it will create a view with all attributes and a selected set of methods (see Figure 9.4 for a default view for an *idm\_hip\_roof* class). Buttons and pull-down menus can be specified in the class template, allowing class methods to be

invoked from the object view independently from the application that created and is manipulating the object. An object view can be created by accessing an extended menu item in the LPA Prolog environment, or through a hot-key. By default, the view command searches the currently highlighted text and creates an object view for every object identifier found in the highlighted text. A complete description of the ObjectViewer can be found in Appendix C.5.

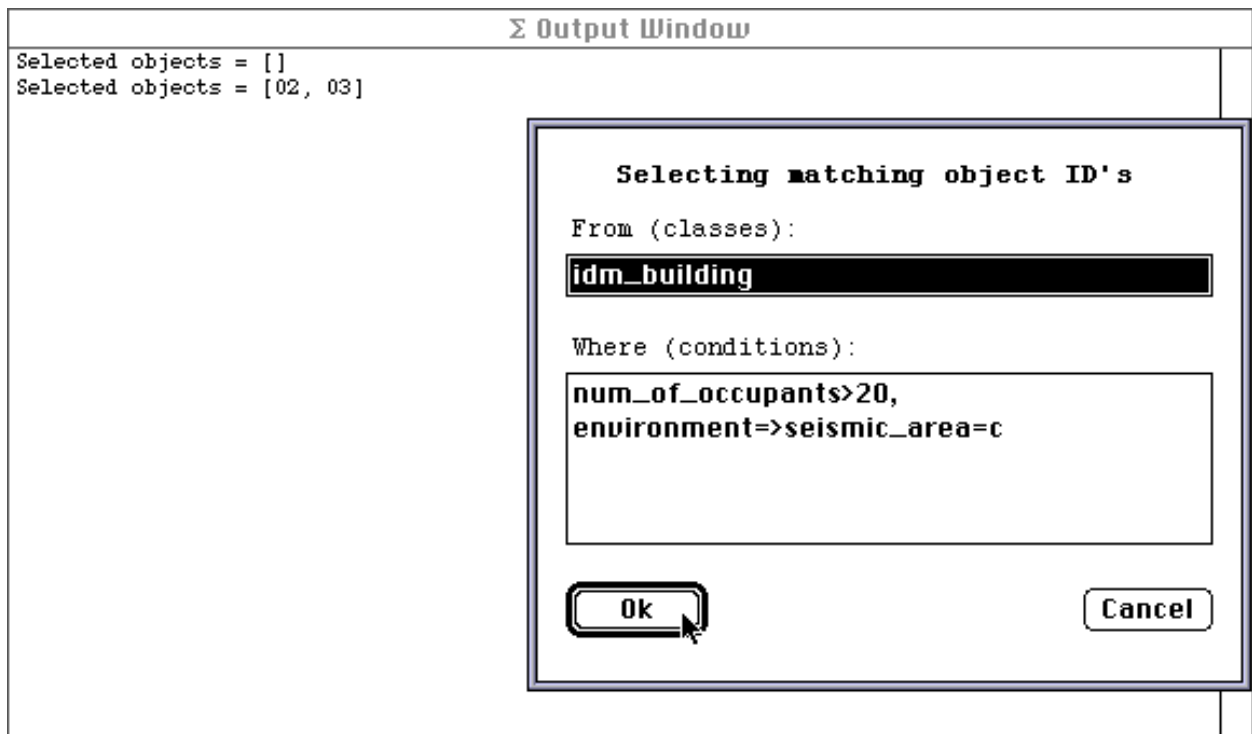


Figure 9.3 A SmartQuery dialogue and result

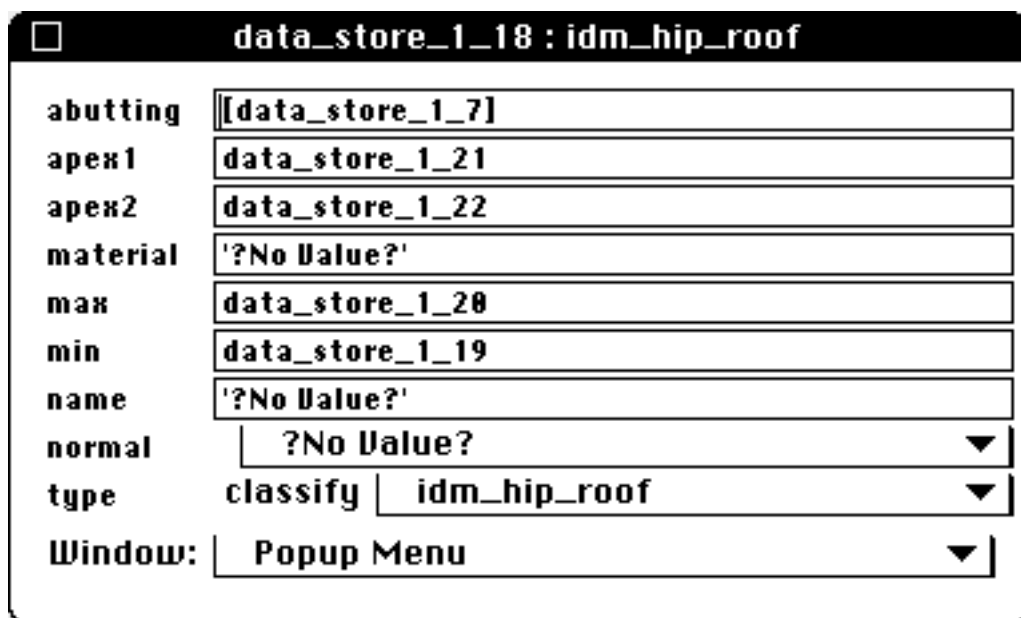


Figure 9.4 A default ObjectViewer object layout

While the ObjectViewer provides direct manipulation of objects and references, along with design tool independent method invocation, it is exactly like EPE in that it does not allow for graphical representation of objects which have an underlying graphical state. The advantage of this tool over

EPE is that it is built directly into the Snart language as a very small and efficient sub-system, unlike the EPE system which is a large system built using Snart, creating a large overhead to have loaded and operating for many applications. The SnartQuery language is also built into Snart and the result of a query is in the form required by the ObjectViewer to enable the identified objects to be viewed.

### 9.2.4 Reflex: an object-oriented CAD system

The commercial OO-CAD system Reflex (Reflex 1996) only became available at a very late stage of the project, but was incorporated as it offers many features which make it highly suitable as an instance creation and manipulation system. Reflex provides the majority of features found in normal CAD systems, with the benefit of a central object-based model. This means that objects with a graphical representation can be quickly created and assembled with other objects. The Reflex system also provides an object and dialogue specification language. This means that new libraries of object types can be created easily. Appendix F.1.3 describes the translator built on top of the EXPRESS parser which allows any schema to be transformed into the required form for a Reflex library. Because Reflex's main domain is buildings, libraries of intelligent object definitions are a standard part. With a small modification of the translated EXPRESS schema it is possible for the library of object definitions to inherit the full specification of the existing Reflex objects. This provides objects which have the complete graphical representation and checking methods as in Reflex, but with the attributes defined in the EXPRESS schema. Figure 9.5 shows a model being developed in Reflex with object definitions translated from an EXPRESS schema. Object definitions can also include default values, which means that complete model definitions can be defined very quickly simply by placing objects in a 2D or 3D Reflex view.

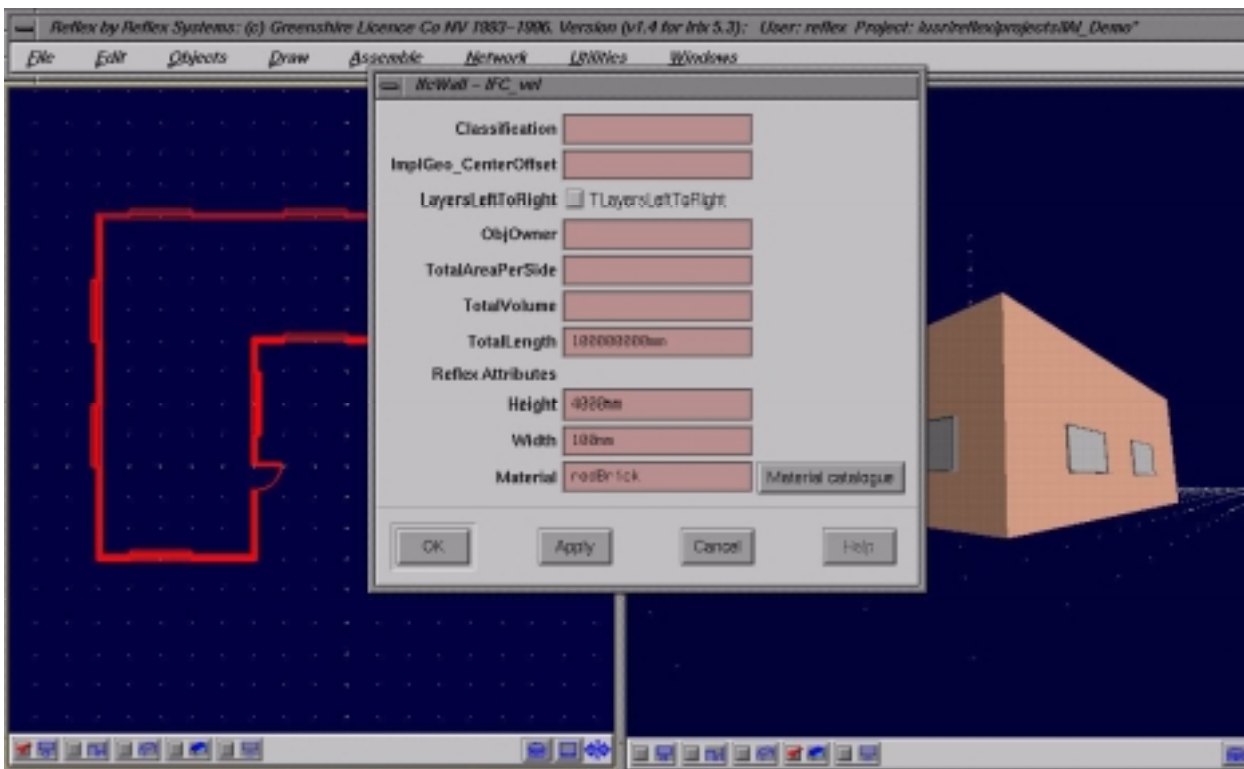


Figure 9.5 Reflex's multiple graphical views, along with an object's attribute dialogue

STEP data-file models can be imported into, or exported from, *Reflex*. Importing models with geometric specifications requires extra coding, as the internal representation of an object's geometry in *Reflex* is different from that specified by all other schemas. In the context of this thesis, *Reflex* is best suited to the initial creation of a model for a particular schema, which can then be exported for use with the other tools described above. Another difficulty is that the relationships between objects that *Reflex* maintains are usually different from those required in the loaded schemas. To handle this, either the *Reflex* object library definitions need to be augmented with code which maintains relationships as defined in the schema, or these relationships must be created and maintained in another tool. The latter method was the one used in this project.

### **9.3 Appraisal of Schema Instance Management**

The set of tools described in Section 9.2 meet the majority of requirements defined in Section 9.1. All of the tools in their standard form can work with any schema definition, requiring no tailoring. However, most can be enhanced by providing further information about the schema definitions. In the case of *Reflex* this provides objects with their most suitable graphical representation and behaviour. With *InSTEP* it allows a model's objects to be rendered graphically if the geometric representation is defined. Multiple views of models are supported, offering geometrical representations, textual representations and graphical layouts of text using *EPE*. These views provide many ways of navigating around instances or through following references, traversing geometric connections between objects, or direct queries to identify required objects. All tools reflect the state of the central model in their views, and *EPE* allows multiple overlapping views of data to be displayed and manipulated.

The only aspect which is not closely managed by any of these tools is the correspondence with an evolving schema. All of the tools except *InSTEP* will allow an internally stored model to be viewed and manipulated with an updated schema definition. However, they will all disregard data values in attributes which don't exist in the new schema, and there will be no checking that attributes which do transfer across match the type now specified for the attribute. All tools make the assumption that in order to move data models to a new version of a schema, a mapping is specified between the two schema versions and the data mapped across before being reloaded into the instance management tools. An extension to the integrated framework described in this thesis could alleviate this problem. The *EPE* tool already tracks all modifications to an evolving schema, a small extension to this tool would enable it to aggregate these modifications together to form the bases of a VML mapping between two states of the schema. Though not all mappings could be captured automatically it would be possible to represent the vast majority of changes with this extension.