Providing CAD Object Management Services through a Base Class Library

HP PE/SolidDesigner's data structure manager makes it possible to save a complex 3D solid model and load it from file systems and databases. Using the concepts of transactions and bulletin boards, it keeps track of changes to a model, implements an undo operation, and notifies external applications of changes.

by Claus Brod and Max R. Kublin

A solid 3D model is a highly complex data structure consisting of a large number of objects. The modeling process requires flexible, fast, reliable, and generic means for manipulating this structure. It must be possible to save the data structure to and load it from file systems and databases. Furthermore, application suppliers need versatile interfaces for communication between the modeling kernel and the applications.

This article describes how the requirements of the solid modeling process translate into requirements for a CAD object manager, and how HP PE/SolidDesigner's *data structure manager* (DSM) is designed to meet these needs.

Besides data abstractions and powerful tools for debugging networks of data, DSM provides a basic data object, the *entity*. An entity's functionality is used by the *entity manager* to file, copy, and scan nets of entities. The *cluster manager* module adds capabilities for building subnets within the whole data structure (*clusters*) and manipulating them. This makes it possible to slice the model into manageable packages that can be sent around the world to subcontractors for distributed modeling. The *state manager* implements a transaction mechanism, which allows the user to browse through the modeling steps and undo changes to the model at any time.

The DSM compares quite nicely with today's object-oriented databases and implements most of their features without the overhead that is often associated with them.

Requirements for a CAD Object Manager

A CAD object manager provides the data infrastructure for the CAD system. It is used by the other components to build and change the model. At the same time, it is a base class library for internal and external programmers. It must fulfill many different user requirements.

It must be able to handle extremely large and complex data structures. When there is a choice of algorithms, the algorithm with the best behavior for large data sets must be selected.

A typical modeling operation changes many individual objects and the structure of the model. Each such change involves the object manager, so its operations will be called very often. Their overhead must be kept at a minimum to prevent the object manager from becoming the performance bottleneck of the system.

Because of the large number of objects, it is also essential that the object manager add only marginal overhead in terms of additional memory to each object.

In a CAD model, many kinds of connections between objects are needed. The object manager should allow and support not only the types of connections that the core product needs, but also any other kind of connection that third-party applications or future modules may require.

CAD programs are large projects which are developed over several years and evolve with the customers' needs. Not all of these needs can be anticipated in the original design. Therefore, the object manager must be flexible enough to allow later extensions, both unlimited new connectivity and completely new kinds of objects. The latter requirement is also essential for third-party applications.

The core solid modeler and its applications operate on the same model. The object manager must offer both sides a view of the model and inform external applications about changes in a generic way. Therefore, the object manager must offer communication mechanisms and interfaces to applications.

The object manager's services are used when building a new type of object and dealing with it. The developer of such a new object will appreciate every kind of support that the object manager can provide, such as debugging tools, handy utilities for frequent tasks, or a library of commonly needed basic data structures, such as lists, tables, stacks and nets.

Finally, the object manager must provide generic mechanisms to store objects and whole models to a file system or database and to load models from there, that is, it has to make the objects persistent.

The design and the use model of HP PE/SolidDesigner add some special requirements to those just described. To support later extensions and the general concept of openness, it is essential that existing object schemes be able to evolve while remaining fully compatible with old data. Furthermore, the object manager, or data structure manager (DSM) in HP PE/SolidDesigner terminology, must support a transaction concept. Transactions must be freely definable

to allow modeling steps that the user perceives as natural. The data structure manager must record all changes to the model in a transaction to be able to roll them back in an undo operation.

The DSM must help to ensure model consistency even if errors occur internally or in external applications. The transaction mechanism can be used to this end.

Concurrent engineering is becoming more and more important in computer-aided design. Files have to be exchanged. Parts of the model are developed independently and assembled later. The data structure manager must support assemblies of parts and the exchange of parts.

Design Principles

HP PE/SolidDesigner's data structure manager was designed with both the above list of requirements and some architectural principles in mind.

One of HP PE/SolidDesigner's key principles is to offer a highly dynamic system with very few static restrictions. The DSM has to support not only today's models, but also future models, so there should be no fixed limits on the size or number of objects. Additionally, the DSM must offer mechanisms to define new objects and object types at run time. This is especially important for external applications.

Each object should only know about its direct neighbors, not about the overall structure of the model. Special data managers are used to collect the local knowledge and form a global picture. This reduces interdependencies between objects which would make later extensions a daunting and dangerous task.

The sequence in which DSM's algorithms traverse the model is not fixed. Since the objects cannot and do not rely on fixed sequences, DSM can also employ parallel algorithms if they are needed and are supported by the hardware and operating system.

Problems in the data structure or in object behavior must be detected as early as possible. In its debug version, DSM checks the consistency of the model thoroughly and offers advanced debugging mechanisms to support the programmer. In the version shipped to the customer (the production version), DSM still employs robust algorithms, but relinquishes debug messages and the more elaborate tests for optimum performance.

Basic Data Abstractions

One way to look at the data structure manager is as a programmer's toolbox. As such, it provides all common building block classes:

- Dynamic arrays
- Lists including ring lists
- Stacks
- Hash tables
- Dictionaries such as string tables and address translation tables
- Bit sets
- Vectors, matrices, and transformations
- Events
- General networks of objects.

These building blocks can be combined to form real-world programming objects. They share basic functionality to standardize their manipulation, such as functions to load and store them, or to scan the data structure and apply a method to each of its elements.

The most important data structure in HP PE/SolidDesigner is the general network. DSM provides net node objects and a net manager class. Each node maintains a list of neighbors in the net. To obtain information about the network as a whole, the net manager visits each individual node, calls its local scan function to retrieve a list of neighbors, and proceeds with the neighbors until all nodes in the net have been visited.

DSM Object Management

The core of DSM is formed by the definition of a generic object, or *entity*, and manager classes that deal with various aspects of entity administration, delivering higher-level services. In the following, we will outline the DSM entity services, beginning with the definition of an entity.

Entities are nodes in a complex network. As such, they use the network functionality described earlier. Additionally, specific entity functions deliver the basic services for transaction handling, filing, object copying, run-time type information, and others.

To benefit from the DSM services, a programmer simply derives a new object from the entity base classes and fills in a few obligatory functions. Almost every object in an HP PE/SolidDesigner model is an entity.

Entities provide a method for inquiring their type at run time. The type can be used to check if certain operations are legal or necessary for a given entity. Object-oriented software should try to minimize these cases, but it cannot completely do without them. An HP PE/SolidDesigner model is an inhomogeneous network of entities. When scanning the net, one finds all kinds of entities. The algorithm that inspects the net often applies to specific types of entities and ignores others. But to ignore entities that we are not interested in, we must be able to check each entity's type.

In an ideal world, type checks could be avoided by using virtual functions. However, to provide these in the base class, it would be necessary to anticipate the functionality of derived classes before they have been created, including those that come from third parties as add-ons to the product.

Run-time type information has been under discussion for a long time in the C++ community, and is only now becoming part of the standard. Therefore, we had to develop our own run-time type system with the following features:

- No memory overhead for the individual object
- Very fast type check
- Checks for both identical and derived types
- Registration of new entity types at run time.

A pure entity is a very useful thing, but certain types of entities are needed so often that we implemented not only one base class, but also a set of standard entities which offer certain additional functionality.

Standard Entity Types

The three most important standard entities are attributes, relations, and refcount entities. *Attributes* are attached to other entities and maintain bidirectional links to them automatically, so they save the user a lot of housekeeping work. For any given type of attribute, only one instance can be attached to an entity. A typical example is the face color attribute. If a face already has been marked as green by a color attribute, attaching another color attribute, say red, will automatically delete the old attribute.

Relations are like attributes, but without the "one instance of each type" restriction. One of the many applications is for annotation texts.

Attributes and relations often are the entity types of choice for a third-party module. They can be attached to entities in the HP/PE SolidDesigner core, and even though the core doesn't have the faintest idea what their purpose is, the connectivity will be maintained correctly through all kinds of entity and entity manager operations. We also use this technique in HP/PE SolidDesigner itself. The 3D graphics module, for example, calculates the graphical representation for the kernel model and then attaches the result to the kernel model as attributes.

Refcount entities maintain a reference counter. Other entities that have a reference or pointer to a refcount entity "acquire" it. Only after the last owner of a refcount entity is deleted is the refcount entity destroyed. (You can think of refcount entities as the equivalent of a hard link in a file system.) Refcount entities can be used to share entities in the entity network to improve memory utilization and performance. We use this type of entity extensively for HP/PE SolidDesigner's geometry.

Nearly all objects in HP PE/SolidDesigner are entities, derived from a common base class. Currently, there are more than 600 different entity types in HP PE/SolidDesigner. Being derived from a common base class, they inherit a set of generic functions which can be applied to any of these 600 different entity types. The most important of these functions are create, delete, copy, store, load, and scan.

HP PE/SolidDesigner allows loading third-party modules at run time. Completely new entity classes can be integrated into the system dynamically. Thus, third-party applications can implement their own entity classes. Entities in external modules are not restricted in any way compared to entities in the HP PE/SolidDesigner kernel. External entities integrate seamlessly into the existing entity network and share all the entity services provided by DSM.

The Entity Manager

In HP PE/SolidDesigner, entities can have any type of connection to other entities. A 3D body, for example, is a very complex network consisting of dozens of entity types. In the entity network of a body, there are substructures such as lists, ring lists, and trees of entities.

An assembly in HP PE/SolidDesigner is a network of other assemblies or subassemblies and 3D solids (parts). This creates another level of structure, in this case a directed, acyclic graph of entity networks.

Suppose we want to copy a part. To do that, we (1) find all entities that belong to the part, (2) copy each single entity, and (3) fix up any pointers in the copied entities. Fig. 1 shows what happens to two entities E1 and E2 that have pointers to each other. First, the entities are copied. In a separate step, the connectivity is fixed. This must be a separate step because when E1c is created (assuming that E1 is copied first), we do not know yet where (at which address) the copy E2c of E2 will be.

Copying a network of entities in HP PE/SolidDesigner is a recurring, nontrivial task. One has to be aware that we deal with dynamic and inhomogeneous networks with entities in them that we might never have seen before because they have been added to the system by a third-party module.

For copying and other entity network services, HP PE/SolidDesigner uses *manager classes*. The entity manager class is an example of a manager class.

Copying an Entity Network

How does an entity manager implement the three steps in copying a part? Step 1 (see Fig. 1) is to find all entities that belong to the part or network. The entity manager only knows that it deals with an inhomogeneous network of arbitrary entities (potentially of unknown type). To find all the entities in a network, the entity manager needs some information about the structure of the network. It collects this information by asking each entity about its direct neighbors in the structure. Suppose the entity manager starts with entity E1. E1 will tell it, "My neighbor is E2." The entity manager will then ask E2 the same question, and the answer will be, "My neighbor is E1." Then—oops, we had better stop here or we will fall into an endless loop! So we see that the entity manager also has to remember which entities in the network it already has visited.

How can the entity manager ask an entity a question, and how can the entity give an answer? The entity manager calls a function (method) called *scan*. Each entity class in HP PE/SolidDesigner provides such a function. We also call this function a *local scanner*. The philosophy behind this is that each entity has a local context, that is, it knows its direct neighbors since it has pointers to them. The entity manager uses this local knowledge of the entities to move forward in a network of entities from one entity to the other, at the same time making sure that each entity will be visited

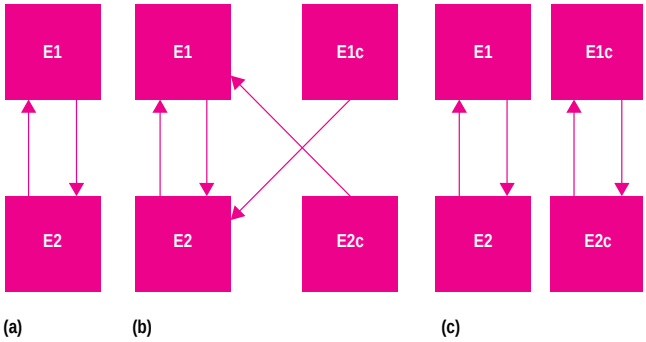

Fig. 1. Steps in copying two entities that have pointers to each other. (a) Before copying. (b) After copying. (c) After pointer conversion.

only once. This we call *global scanning*, and it is implemented in the entity manager's scan function.

The restriction that each entity in the network is only visited once becomes really important only if a certain operation has to be executed on each entity. Therefore, the entity manager's scan function not only receives a start node (the entry point into the network), but also a *task function*, which is called for each node that is visited in the network.

With the knowledge gained from scanning the network, we can move to step 2, copying each entity. The task function that is passed as a parameter to the entity manager's scan method solves this part of the problem by calling the *copy* method of each entity. This is another method that every entity in the system provides.

While in step 2, we have to make provisions for the next step. We record in a table where each entity has been copied to. For each entity, the task function creates an entry of the form [old entity address, address of the copy] in this table. Actually, this table is a hash table that can be accessed using the old entity address as the key. Address translation tables like this are used in many other places in HP PE/SolidDesigner, so DSM offers a special pointer dictionary class for this purpose.

After step 2, we have a copy of each entity and we have built an address translation dictionary. Now we're ready for step 3. For each entity in our dictionary, or more precisely for each entity recorded in the right side of a dictionary entry, we call another method, convert pointers. By calling the convert pointers method, we request that the entity convert all the pointers it has local knowledge of. In the case of the entity E1c (the copy of E1), for example, this means, "I have an old pointer to E2, and I need to know where the copy of E2 (E2c) is." This question can be answered using the address translation dictionary built in step 2 since it has an entry of the form [E2, E2c] in it. After we have called the convert pointers method for each copied entity, we are finished. We have copied a network of entities without knowing any of these entities!

So far, so good. Now we know how to copy a network of entities in main memory. At some point, the entities will have to wander from main memory to permanent storage. Therefore, let us examine next how we store and load a network of entities into and from a file.

Storing and Loading an Entity Network

Storing and loading, like copying, are operations on a network of entities. Therefore, the entity manager provides these functions. Storing a network of entities works like this:

(1) Open a file.
(2) Find all entities that belong to the network.
(3) For each entity:
    (a) write an entity header
    (b) store the entity
    (c) write an entity trailer.
(4) Close the file.

Besides opening and closing the file, storing essentially means writing each entity in the network into a file. This sounds simple enough. To solve the problem, we can even use existing functionality. The entity manager's scan method will help us find all entities in a network, just as it did for copying.

All we have to do is to provide a new task function which executes step 3 for each entity. In 3a and 3c we write administrative information that we will need for loading. For 3b we need a way to store an entity generically. Of course, we want not only to store, but also to load entities. Therefore, each entity has a store method and a load method. The store method is an ordinary member function of the object. The load method, however, is a static member function since it creates the object out of the blue (well, actually, from the information in the file) and then returns it.

When everything is stored, the file contains entities in a form that is equivalent to the situation in step 2 in the entity copy operation. All pointers between entities are invalid, and they have to be fixed when the file is loaded again.

Loading a file is also a task for the entity manager, since it deals with a whole network of entities. Loading works as follows:

(1) Open the file.
(2) While not at the end of the file:
    (a) read the entity header
    (b) call the entity's load method (a new entity is created in main memory)
    (c) enter the entity information into a dictionary
    (d) read the entity trailer.
(3) Close the file.
(4) For each entity in the dictionary, call the convert pointers method.

Reading the Entity Header. The entity header contains two important data items: the entity type and a virtual address. The entity manager uses the entity type to decide which of the 600 or more different load functions is to be called. When storing an entity, the object exists and its store method can be called. When loading entities, a different approach must be taken. The entity manager maintains an entity type table which can be added to dynamically. For each entity, the table contains, among other things, a load function.

Note that an entity type translates into a class in C++. All objects of a class have the same type (for example, face).

The second data item in the header is the *virtual entity address*. The virtual address is a unique entity ID which is used to represent pointers between entities in the file. When storing an entity, the entity does not know where a neighbor entity that it points to will be placed when the file is loaded again. Therefore, all pointers between entities in the file are virtual pointers and have to be converted after loading the file.

Calling the Load Method. The entity manager detects the type of the entity from the entity header. It will then call the

Exception Handling and Development Support

DSM has its roots in the late eighties—the early days of C++. Compilers didn't support exception handling then. Conventional error handling by passing error codes up the return stack is a prohibitively code-intensive approach in a large software project with many nested procedural levels such as HP PE/SolidDesigner. Therefore, we had to implement our own exception handling mechanism which is very similar to what has been implemented in today's C++ compilers.

HP PE/SolidDesigner's code is divided into code modules. Each module has its own *module information object* containing module-specific error codes and messages. In case of an error condition inside a module, the code triggers the exception mechanism by throwing a pointer to the module information object.

Code that wants to catch an exception inspects the module information object returned by the exception mechanism and acts accordingly. If it has already allocated resources, they are cleaned up and returned. The exception can then be ignored (and suppressed), or it can be escalated to the next code level.

The listing below shows a code example for this. You may notice the similarities to the exception handling mechanism introduced with C++ 3.0. Now that the throw/catch mechanism is finally available in many C++ compilers on various platforms, we will be able to adopt it with only a few changes in the code.

```
int process_file(const char *const fname)
{
  int words = 0;
  FILE *file = 0;

  TRY
    file = open_file(fname);
    words = count_words(file);
    close_file(file);
    file = 0;
  RECOVER
    if (file) {                                    // clean up resources
      close_file(file);
      file = 0;
    }

    // handle specific exceptions
    if (dsm_exception_code == F2_CORE::info_ptr) {
      switch(F2_CORE::errno) {
    case F2_CORE::BREAK_RECEIVED:     // User has cancelled processing
        // We won't escalate this "soft" exception.
        handle_break();
        break;
     case F2_CORE::MEM_OVL:                    // Out of memory
        // Free memory blocks allocated here, then escalate the problem.
        free_my_mem();
        ESCAPE(dsm_exception_code);             // "throw" in C++ 3.0
        break;
      default:
        break;
      }
    } else {
      // Pass up all other exceptions.
      ESCAPE(dsm_exception_code);
    }

  END_TRY

  return words;
}
```

Development Support

To find problems proactively, DSM stresses the importance of checking preconditions, invariants, and postconditions. It offers convenient assertion macros and a context dependent run-time debugging system which uses *debug module objects*.

These debug module objects hold their current debug level which can be checked using macros and set during run time. A debug module is associated with a certain code area. This allows fine-grained control for debug checks and messages. We think this control is important for the acceptance of a debug system; the programmer will ignore debug messages if there are too many, and won't find the system useful if it doesn't deliver enough detail where needed.

Macros are provided to reduce typing and #ifdef constructs:

```
bool compare(const char *s1, const char *s2)
{
  ME_MODULE_STOPWATCH("compare", foo);      // for run-time profiling

                                            // trace program flow
  if (DEBUG_LEVEL(foo) >= DEBUG_CALLS) {
    fprintf(DEBUG_STREAM(), "compare called");
  }

  DSM_ASSERT(s1 && s2);                      // check precondition

  // Now calculate the result
  ...

  DSM_ASSERT(some_condition);                // check post-condition
  return TRUE;
}
```

DSM also defines special debug modules to switch on sophisticated debugging tools. There are tools to find memory leaks, to calculate checksums for objects (allowing us to detect illegal changes), and to create run-time profiles for the code.

In a software package as large as HP PE/SolidDesigner, the common UNIX profiling tools were not applicable. Therefore, we had to build our own set of versatile, efficient and highly precise utilities. You can define a *stopwatch* for any function that might need profiling, and you start and stop the stopwatch using the debug module mechanism. The results can be analyzed, producing a hierarchical call graph that shows what portion of the run time was spent in the individual functions. We can also find out the amount of memory allocated for a function at run time using these tools.

right load function, using the information in its type table. This transfers the control to the entity's load method which is responsible for creating a new entity from the data in the file. The new entity is returned to the entity manager. Creating an entity from a given type implements a virtual constructor function, which is missing as a language element in C++.

Entering the New Entity into a Dictionary. Here we create an entry in a dictionary that contains the virtual entity address in the file and the new real address in main memory. These values will be used in pointer conversion.

Reading the Entity Trailer. When the entity is loaded, the entity manager resumes control by reading the entity trailer. This might appear to be an artificial overhead operation, but it makes sense when we consider the dynamic nature of the system. We mentioned earlier that new entity types can be created and registered dynamically, for example by a third-party module. When storing an entity network, these entities

are also stored. A user might try to load such a file into an HP PE/SolidDesigner system that does not know about these entities because the third-party module has not been installed. When the entity manager loads such a file, it will encounter entity headers of entity types for which a load function has not been announced. Here's where the entity trailer helps. The entity manager simply skips all following data in the file until it finds the entity trailer. Thus, HP PE/SolidDesigner ignores unknown entities in a file, but it can still load the rest of the file.

Converting Pointers. After loading, all pointers between entities are virtual and have to be converted into real memory addresses. For each entity in the dictionary, that is, for each entity that has been loaded, its convert pointers method is called. We have already discussed this method for copying networks of entities. Each entity knows its pointers to other entities, and it asks the entity manager, "Now I have a virtual pointer to entity E1, so please tell me where E1 is in main memory." For each pointer, the entity calls the entity manager's convert pointer service function. This function is passed a virtual entity address and returns the real memory address of the loaded entity. The dictionary built while loading the file contains the necessary information.

When all entities have been converted, we have written a network of entities into a file and loaded it from there without knowing any of the entities in detail. The analogy to the copy operation does not come by chance, but is the result of careful design. For copying or storing and loading entity networks, DSM employs the same functionality wherever possible. In theory, we could have built the copy operation completely on a store and a subsequent load operation.

Entity Revisions

As the CAD system evolves, the need arises for changes in entity layout, either by adding a new data field or by changing the meaning of an existing one. In object database terms, this is known as the schema evolution problem. The load function of a DSM entity can check the revision of the entity in the file before actually loading the contents of the entity. Depending on the entity revision, the load function will then know what data fields are to be expected in the input. This means that the load function is prepared for *any* revision of the entity. The same holds true for the store function, which can write different revisions of an entity depending on the given storage revision.

This feature ensures upward compatibility of HP PE/SolidDesigner files. All new versions automatically know about the old object revisions, and no converters are necessary. In database language, our object database can be inhomogeneous with respect to entity revisions. From a pure DSM point of view, even downward compatibility is possible, since you can set the storage revision to a previous level and then save a model, as long as the new revision did not introduce new entities that are essential for the overall consistency of the model in the new scheme.

The Cluster Manager

From the entity manager's point of view, the current HP PE/SolidDesigner data model is one coherent network of entities. Each and every entity will be reached when the entity manager's global scan method is used. The user's point of view, however, is different. The user works with well-defined objects such as parts, workplanes, assemblies,workplane sets, layouts and so on, which can be arranged in a hierarchy. An assembly is like a directory in a file system, and a part is like a regular file. Assemblies can have subassemblies just as directories can have subdirectories, and parts and assemblies can be shared just as directories and files can be linked in a file system.

The cluster manager closes this gap between the entity world and the user's perception. It creates facilities to define a *cluster* of entities—for example, all entities that belong to a part. There is no hard-coded knowledge about cluster structures in the cluster manager, however. Instead, the entities in the network themselves define what the cluster is. Because of this flexibility, the cluster manager can offer its services for any kind of entity network.

The following algorithm collects all entities belonging to a given cluster X:
(1) Start with a representative of the cluster and look for all direct neighbor entities.
(2) Ask each entity found during the scanning process to which cluster it belongs.

(a) If the entity's answer is "I belong to cluster X," continue the search with the entity's neighbors.
(b) If the entity answers "I belong to cluster Y," the global search has arrived at a cluster boundary. The entity is excluded and the search will not be continued from this point.

The entity manager's scan method helps with (1), and the cluster manager provides a task function for (2). The task function's return value controls how the entity manager navigates through the network of entities. It is the entity manager's job to find the neighbors for each entity and to ensure that nodes are visited at most once.

There are implications for the topology of a cluster: it must be possible to reach any entity in the cluster using a path that is completely within the cluster. Figs. 2 and 3 show examples of correct and malformed clusters.

How can an entity tell to which cluster it belongs? Actually, this is asking too much of a mere entity. What we can expect from an entity, however, is that it can point us in the direction of another entity that is one step closer to the representative of the cluster. Each entity has a *local master* method for this purpose.

In most cases, the entity chooses one of its neighbors as its local master, but this is not obligatory. By following the trace laid out by the individual local master functions, we will eventually find the main representative of the cluster (which is special in that it points to itself when asked for its local master). We call this special entity the *cluster master*.

Note that this is another case in which we build global knowledge from local knowledge at the individual entities. This is how we can define a cluster structure in a complex network. The highlights of this method are:

- The entity manager's global scanning services are used.
- The entities need local context only.
- Only one additional method, local master, is needed for each entity.

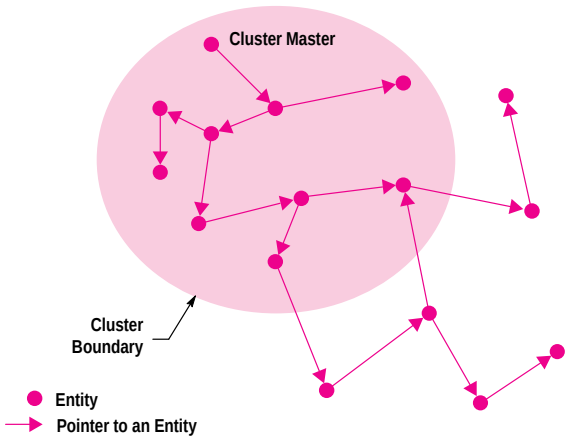

Fig. 2. A correct cluster.

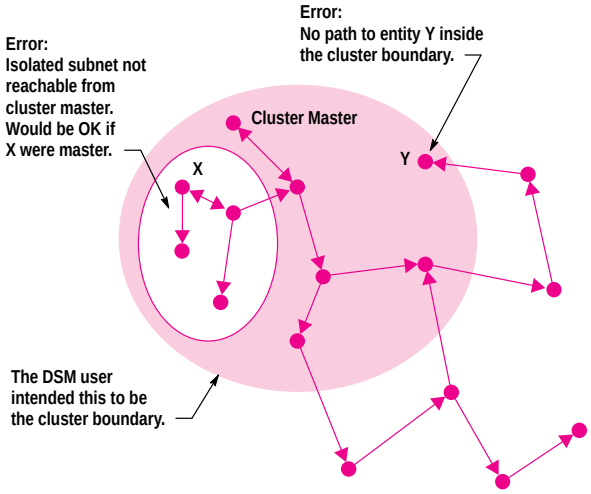

Fig. 3. An illegal cluster.

- The approach is fully object-oriented. The objects themselves determine the size, structure, and shape of the cluster. Completely new entities can be integrated into the cluster in the future, and completely new clusters can be built.

The cluster manager offers services for storing, loading, and copying clusters. It implements these by using the entity manager's basic services. The entity manager is controlled by cluster manager task functions, which determine the (cluster) scope of each operation.

The cluster manager services can be used to handle an individual part or a workplane. The cluster manager also supports hierarchical structures such as assemblies and workplane sets.

Fig. 4 shows two types of screwdrivers. They share the shaft; only the blades are different. The parts browser shows the part hierarchy. The notation "(P :2)" indicates a shared part and the backward arrow "<–" indicates the active part (which is also highlighted in green). The shaft part is contained in both assemblies. When using standard parts, we will in fact by default have many instances of the same part (or even whole assemblies) in multiple assemblies. If we now change something in the shared part (in this case the shaft), we expect the changes to be reflected in both assemblies, since both assemblies have a reference to the same part. This we call *sharing parts and assemblies*. Workplanes can also be shared by using them in different workplane sets.

In the base version, HP PE/SolidDesigner stores the model data to files in the regular file system. To ensure that the sharing is preserved when storing and loading models, the following rules apply:

- Every object that can be shared in HP PE/SolidDesigner has its own file in the file system.

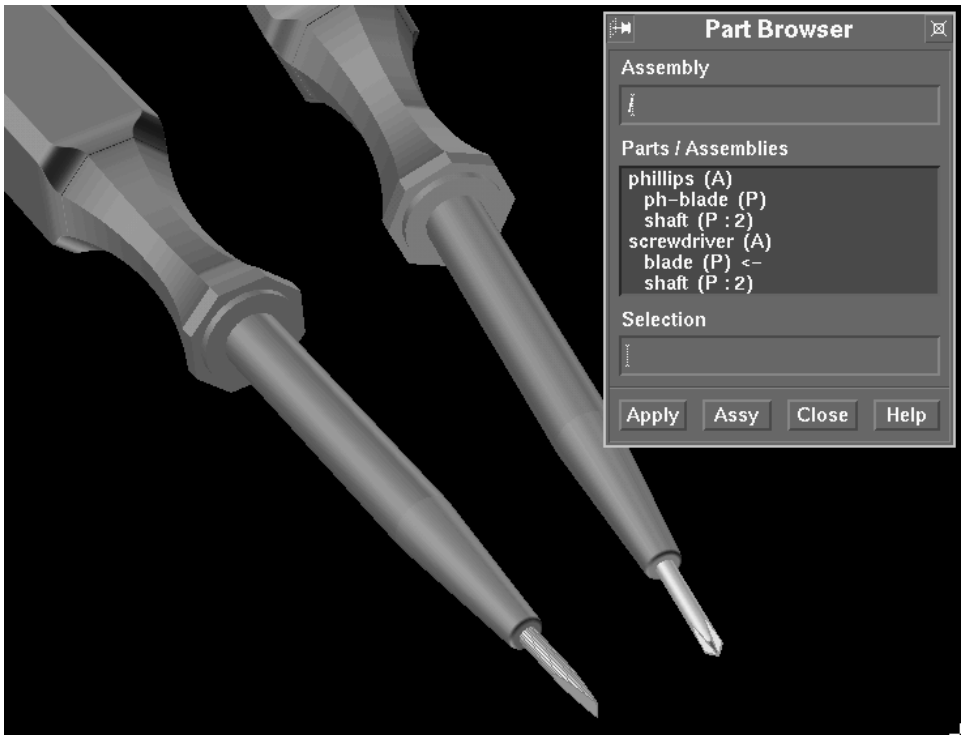

Fig. 4. Two assemblies with shared parts.

- For a shared object, exactly one file exists, regardless of how many owners the object has. This makes sure that whenever the shared object changes, all instances will be changed as well.
- When storing an assembly, all objects below the assembly have to be stored as well. This ensures that the data in the file system is complete, so that another HP PE/SolidDesigner system can pick it up immediately.
- A file contains exactly those entities that correspond to one cluster.

Suppose we want to store the screwdriver assembly. We expect that three files will be created: one for the assembly, one for the blade, and one for the shaft. The cluster manager will do this for us; we just tell it to store the screwdriver assembly. It will find the parts and any subassemblies of the assembly on its own. Since the cluster manager must work with an arbitrary network, it needs another entity method, *scan child clusters*, to build on. This method is implemented by those (few) entities that take over the role of a cluster master. The scan method of each entity would not help us here since it just gives us access to all direct neighbors without helping us determine a direction.

The cluster manager uses the scan child clusters method to find the children of a cluster in a generic way. Applying the method recursively, all objects within the assembly can be found. It is possible that a child will be reached more than once (for instance, a standard screw within a motor assembly). The cluster manager keeps track of the clusters that have already been visited to prevent a cluster from being stored twice.

Given these methods, we can describe how an assembly (actually, any kind of cluster structure) is stored:
- Start with the given cluster and find all children recursively.
- For each child cluster, use the entity manager's store method to store the entities of the cluster into a separate file. The entity manager is controlled by a cluster manager task function that makes sure that only those entities belonging to the cluster are stored. A special store pointer function is responsible for storing pointers to entities.

The store pointer function deserves a discussion of its own. When storing clusters into several separate files, we will encounter pointers that point from one cluster (file) to another. In the case of the screwdriver assembly, we will have at least two pointers to the external clusters representing the blade and the shaft. Since the entity manager's store function by default stores all entities in the network into one file, the problem doesn't arise there. By providing a special store pointer function, the cluster manager extends the entity manager so that pointers are classified as *external* (pointing to another file) or *internal* when they are stored.

When loading an assembly, the cluster managers goes through the following procedure:
(1) Open the file.
(2) Use the entity manager's load method (with the special load pointers function) to load all entities in the file.
(3) Close the file.
(4) While there are external references to other clusters left, open the corresponding file and proceed with (2).

An external reference is a pointer to an entity in a different cluster. To make sure that external pointers are unambiguous, we developed a scheme for unique entity IDs. An entity is assigned such an ID when it is created, and it keeps it as long as it exists. External pointers refer to these unique IDs.

The algorithm above is analogous to linking relocatable object files in the HP-UX* operating system. When loading the file into HP PE/SolidDesigner, it is the special load pointer method's job to detect external references. In step (4), the cluster manager behaves quite similarly to an object file linker. Where the linker needs one or more libraries, which it searches for objects to satisfy open references, the cluster manager uses the UNIX® file system or a database as its library.

The State Manager

The state manager introduces a notion of transaction handling into HP PE/SolidDesigner. Model changes can be grouped together to form a single *transaction*. In database technology, a transaction has the following properties:
- Atomicity. The transaction is *atomic*. It must either be closed completely or undone.
- Consistency. Transactions transform a given consistent state of the model into a new state which again must be consistent in itself.
- Isolation. Transactions do not influence each other.
- Durability. The changes made by a transaction cannot be cancelled by the system except by special undo transactions.

Transactions in HP PE/SolidDesigner have these properties. They are not only used for ensuring data integrity, however. Their main purposes in HP PE/SolidDesigner are to notify kernel applications about changes in the model at defined intervals (when a transaction is completed) and to allow interactive undo operations.

The general model of an HP PE/SolidDesigner transaction is shown in Fig. 5. A transaction T12 transforms a given consistent model state S1 into a new consistent state S2. A rollback to S1 is possible. As Fig. 5 shows, it is also possible to roll forward, that is, move towards the modeling "future" after an undo operation.

Bulletin Board

DSM introduces a special mechanism to record changes to the model, which is the *bulletin board*. Information about all changes within a transaction are collected in one bulletin

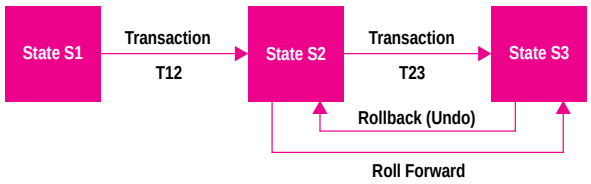

Fig. 5. HP PE/SolidDesigner transaction model. A transaction transforms one state into another. A transaction can be rolled back or rolled forward.

board. In other words, the bulletin board describes the transaction completely, so that we sometimes use "bulletin board" and "transaction" interchangeably.

A bulletin board is a collection of individual *bulletins*. A bulletin describes a change of state of a model entity, that is, it contains delta information. At the beginning of a transaction, the bulletin board is empty. Each change to an entity creates a bulletin describing the change, so at the end of the transaction, the bulletin board contains all of the changes that happened during the transaction.

When a transaction completes, a special event, the *transaction end event*, is triggered. *Update handlers* subscribe to this event. When they are called, they receive as a parameter a pointer to the bulletin board created in the transaction. They can then inspect the contents of the bulletin board to look for changes that they have to act upon. The 3D graphics module, for example, which, slightly simplifying things, is just an update handler, checks for the creation or changes of 3D bodies. It then creates a faceted graphics model from the change information that is suitable for sending to a graphics library. Since it only deals with the delta information, the 3D graphics handler will in general complete its job more quickly than if it regenerated the whole graphics model after each transaction.

An update handler may also choose to ignore the bulletin board information. It will then use the transaction end event as a regular opportunity for cleanup tasks or to rescan the model. Most update handlers, however, use the information in the bulletin board to optimize their work.

Changes

The DSM's state manager module uses basic entity services to create bulletin board information. To provide systemwide transaction handling and the undo mechanism, each entity has to follow a few simple conventions. The most important of these conventions is that *before* any kind of change to itself, an entity has to announce the change. It does so by calling a special *log change* method, which is provided by the entity base classes.

The log change method does a lot of things. First, it creates a bulletin in the bulletin board. The log change method is passed a *change type* from the caller which it also records in the bulletin. Using the change type, the changes are classified, and update handlers can ignore changes of types they are not interested in. They can also ignore changes to certain entity types. Using these two restriction types, update handlers can narrow down the search to a few bulletins even if the transaction is very large.

After building the bulletin, the state manager uses the entity's generic copy method to create a backup copy of the entity. Note that the entity is still in the original state since the log change method has to be called before any change takes place. (To ensure that the convention is followed, we have built extensive debugging tools that detect changes that are not announced properly.)

Pointers to both the entity in its current state and the backup copy of the entity are maintained in the bulletin board. This gives the update handlers a chance to compare the data in an entity before and after the change, making it possible for an update handler to trigger on changes to individual data items in the entity.

So far, we have only discussed changes to an entity. The bulletin board also records creation and deletion information for entities. The entity base classes, together with the state manager, take care of this.

In an undo operation, all changes to entities are reversed. An entity that has been reported as deleted will be recreated, and new entities will be marked as deleted. (They will continue to exist in the system so that it is possible to roll forward again.) If an entity has changes during a transaction, its backup copy will be used to restore the original state. Again, we use the generic copy function in the entity base classes for this purpose.

Relation to Action Routines

The action routines (see article, page 14) define when a transaction starts and ends. When the user selects an operation in the user interface, an action routine will be triggered that guides the user through the selection and specification process. A transaction is started at the beginning of such an action routine. After each significant model change, the action routine completes the transaction, thus triggering the transaction end event and giving update handlers a chance to react to the changes.

When an action routine terminates without error, all transactions generated within the action routine are usually merged into one large transaction. Thus, the user can undo the effect of the action routine in one step. If an error occurs within an action routine, all changes in the action routine will be undone using the generic rollback mechanism and the information in the bulletin boards.

Some action routines also implement *minisessions*. After collecting all the options and values, the operation itself can be triggered and its effect previewed. If the effect is not what the user thought it should be, it can be undone within the action routine. The minisession will then use the rollback mechanism internally. The user changes parameters, triggers the operation again, and finally accepts the outcome when it fits the expectations. An example of this in HP PE/SolidDesigner is the blend action routine.

In general, however, operations can be undone using the interactive undo mechanism. At any point, the user can choose to roll back to a previous state. For this purpose, HP PE/SolidDesigner keeps the last n states (or bulletin boards) in memory where n is a user-configurable value. The user can also move forward again along the line of states that was created in the modeling session.

Fig. 6 shows HP PE/SolidDesigner's user interface for undo operations.

As discussed earlier, HP PE/SolidDesigner's transaction mechanism also offers an interface to external applications, that is, the transaction end event. Third-party applications subscribe to the event, and from then on, they can monitor all changes to the model. One example of an "external" application is the 3D graphics module. Parts browsers, which also have to react to changes of the model, are another example. Finite-element generators can also hook into the

Fig. 6. User interface for undo operations.

transaction end event to keep track of the model. Another possible external application is one that provides the current volume properties of given bodies. (HP PE/SolidDesigner provides volume calculations, but they have to be triggered explicitly from the user interface.) The bulletin board is the door-opener for external applications, making it one of the most important interfaces within HP PE/SolidDesigner.

Conclusion

This article can only give a very high-level overview of what DSM is all about. Much of what really makes DSM usable, effective, and efficient is beyond the scope of this discussion. We are confident that the data structure manager is a strong and robust building block for any kind of application that has to deal with complex data networks. We have found that DSM deals with a lot of problems that are typical for object databases:

- Data abstraction (through a set of base classes)
- Object persistence (storing and loading objects)
- Object schema evolution (changes in object layouts)
- Object clustering (bundling low-level objects to user-level objects such as parts and assemblies)
- Exchange of clustered objects, fully maintaining connectivity through unique object IDs)
- Transaction concept with undo.

By solving all of these problems, DSM enables HP PE/SolidDesigner to support typical modeling operations on user-level objects (parts, workplanes, etc.). In other words, it makes HP PE/SolidDesigner speak in terms that the user can easily understand. The support for object exchange is the basis for modeling workflow solutions. Apart from this, the data structure manager can serve as a general framework for any kind of object-oriented application.

Acknowledgments

The data structure manager was initially designed and developed by Peter Ernst. He is still our sparring partner for discussing new ideas and the general direction of development for DSM.