HEWLETT-PACKARD
JOURNAL

table of contents

October 1995,
Volume 46, Issue 5

Articles

HP PE/SolidDesigner: Dynamic Modeling for Three-Dimensional Computer-Aided Design

In most solid modeling CAD systems, knowledge of the history of the design is necessary to avoid unanticipated side-effects when making changes. With dynamic modeling, local geometry and topology changes can be made independently of the model creation at any time, using both direct and dimension-driven methods. The core components enabling dynamic modifications are the tool body and the relation solver.

by Klaus-Peter Fahlbusch and Thomas D. Roser

HP Precision Engineering SolidDesigner (PE/SolidDesigner) is a 3D solid modeling design system based on the ACIS® Kernel (see "About Kernels" on next page). It provides the geometric model needed by design workgroups in product development environments. The system's dynamic modeling technology gives the designer the freedom to incorporate changes at any time and at any stage of product development, without dependence on the history of the product design.

HP PE/SolidDesigner is a member of the HP Precision Engineering Systems (PE/Systems) product family. Today, HP PE/Systems consists of:

- HP PE/SolidDesigner for solid modeling
- HP PE/ME10 for 2D design, drafting, and documentation
- HP PE/ME30 for 3D design
- HP PE/SurfaceStyler, an engineering styling application integrated with HP PE/SolidDesigner
- HP PE/SheetAdvisor, a sheet-metal design-for-manufacturability application
- HP PE/WorkManager for product data and workflow management
- HP PE/DDS-C for electrical system design
- HP PE/Complementary Application Program (CAP), a joint research and development and marketing program that provides HP PE/Systems users with access to more than 200 leading applications from 70 companies.

HP PE/SolidDesigner

HP PE/SolidDesigner makes it easy for designers to move to 3D solid modeling. It supports the coexistence of surface data with solid data and provides the ability to import and modify surface and solid design data from a variety of CAD systems. It also offers new modeling functionality and enhanced ease of use.

Using improved IGES (Initial Graphics Exchange Standard) import capability, both surface and wireframe data can be imported. Surface data and solid data can also be imported and exported using the STEP (Standard for the Exchange of Product Model Data) format. Once imported, this data can coexist with HP PE/SolidDesigner solid data. It can be loaded, saved, positioned, caught to (see footnote on page 15), managed as part and assembly structures, deleted, and used to create solids. Attributes such as color can be modified. If the set of surfaces is closed, HP PE/SolidDesigner will create a solid from those surfaces automatically. Other solid modeling systems, which are history-based, are unable to import data and then modify it as if it had been created within the system itself.

HP PE/SolidDesigner allows solid parts and assemblies to be exported to ACIS-based systems using Version 1.5 of the ACIS SAT file format. This feature provides a direct link to other ACIS-based applications.

With HP PE/SolidDesigner, users can set part and layout accuracy. Because users can model with parts of different accuracy by forcing them to a common accuracy, they can import and work on models from other CAD systems regardless of their accuracy.

Dynamic modeling is the underlying methodology within HP PE/SolidDesigner. This flexible, nonhistory-based, intuitive design technique provides direct interaction with modeling tools and designs, allowing the engineer to focus effectively on the design task.

HP PE/SolidDesigner allows designers to work with user-defined features to capture design intent. Users can explicitly group a variety of 3D elements such as faces and edges of a part. These features then can be viewed, edited, renamed, deleted, or used to drive changes to a design.

HP PE/SolidDesigner has variable radius blending, which allows users to create, modify, and remove variable blends. They can now create constant and variable blends during one session. Another new feature, called shelling, provides a quick way for users to create thin-walled parts from solids, as in injection-molded parts, for example.

Also new in HP PE/SolidDesigner is mass property capability. The following properties can be calculated for parts and assemblies: face area, volume, mass, center of gravity, inertia tensor, and boundary area. Tolerances can be supplied and achieved accuracies are returned. HP PE/SolidDesigner also incorporates interference-checking capabilities, which

allow detection of interference, face touching, and noninterference of assemblies and part combinations. The results can be shown as text reports or in graphic format with color coding for easy identification.

About Kernels. A kernel is the heart of a modeling system. Currently, three kernels are used in various CAD systems. These are Romulus from Shape Data, Parasolid, an extension of Romulus, and the ACIS Kernel from Spatial Technology. The ACIS Kernel is rapidly becoming a de facto standard, having been accepted to date by 25 other commercial licensees, 50 academic institutions, and 12 strategic developers. As of July 1995, companies that officially have committed to using ACIS as their underlying technology include MacNeal-Schwendler/Aries, Applicon, Autodesk, Bentley Systems, CADCentre, Hewlett-Packard, Hitachi-Zosen Information Systems, Camax Manufacturing Technologies, Intergraph, and Straessle.

About STEP. The STEP protocol for data exchange is the product of a group of international organizations including PDES/PDES Inc. USA, a joint venture with several member companies, ESPRIT (European Strategic Program for Research and Development in Information Technology), European data exchange technology centers such as CADDETC (CADCAM Data Exchange Technical Centre) and GOSET, and ProSTEP, the German industry project for establishing STEP in the automotive industry.

HP has been active in STEP technology since 1989 through projects such as CADEX (CAD Geometry Exchange), PRODEX (Product Data Exchange), and ProSTEP. HP provides STEP processors with its HP PE/SolidDesigner 3D solid modeling software.

Dynamic Modeling

Currently, the most popular 3D CAD solutions are history-based. When designing with these systems, dimensions and parameters have to be specified at the outset. The model can only be manipulated indirectly by modifying these dimensions and parameters. The initial definitions have a major influence on the ease or difficulty of carrying out subsequent modifications, which can only be reliably implemented if all the previous steps in the design process are known. Laborious manipulation may be necessary to make changes that, intuitively, should be achievable in a single step.

Unless the history of the design is thoroughly understood, any change made to a model may have unanticipated side-effects. Relatively straightforward changes to the model involve many convoluted steps. Future interpretation becomes ever more difficult and the effects of further modifications are unpredictable. Even when a single designer takes a part from start to finish, the designer will usually recreate the model from scratch many times as decisions made earlier make further progress impossible.

Although history-based systems are appropriate for solving family-of-parts problems, and are ideal for companies who simply produce variations on a given design, they are inflexible when used during the conceptualization phase of a project.

Dynamic Modeling

Dynamic modeling has been developed by HP to overcome the many problems designers experience with history-based CAD systems. In particular, it aims to remove any dependencies on history and the need to anticipate future changes.

The concept underlying dynamic modeling is to make optimal use of technologies without constraining the designer's creativity and flexibility. In contrast to history-based systems, dynamic modeling allows direct manipulation of model elements in 3D space. With dynamic modeling, local geometry and topology changes can be made independently of the model creation at any time, using both direct and dimension-driven methods. In the latter case, dimensions can be specified at any stage in the design, not just at the outset.

The core components enabling dynamic modifications are the tool body and the relation solver. To make a model modification a tool body is created and then transformed to the appropriate position. A Boolean operation between the original model and the tool body results in the desired model modification.

HP PE/SolidDesigner is the only currently available CAD solution that uses dynamic modeling. The remainder of this article describes the underlying technology of dynamic modeling and compares it with other methods like parametric model modification techniques.

State of the Art

Currently, solid modelers use two different approaches to create the final geometrical model. CSG (constructive solid geometry) modelers are based on volume set operations with volume primitives such as cubes, cones, or cylinders. This approach is characterized by a Boolean engine, which implements the basic operators unite, subtract, and intersect. The sequence of all the Boolean operations, parameters, and positions of the primitives are kept in the *CSG tree*. Modification of the solid later in the design process can be done by using more primitives or by editing the CSG tree. Local modifications of the model are not possible, since no access to faces or edges is given. This cumbersome way to modify solids requires the user to analyze the design beforehand and dissect it into the necessary primitives and operations. While anticipating design modifications and building designs out of primitives is not typical in the mechanical engineering design process, pure Boolean modelers have proven useful when entering a final design for postprocessing, such as for finite-element analysis (FEM) or NC tool path programming.

B-Rep (boundary representation) modelers represent the solid by concatenating surfaces towards a closed volume. Model creation is similar to CSG modeling, but the user can work locally with surfaces, trim them against each other and "glue" them together. Local geometry modifications are very flexible and represent the way engineers think. For example, "I would like to blend this edge" is a natural way of specifying a model change for a mechanical engineer, while "I have to remove a volume that cuts away all material not needed" is a very unnatural way of specifying the same task during design.

As the development of B-Rep modelers continued, a new class of operations emerged in the early 1980s from the research institutes and appeared in commercial implementations. These operations are called *local operations*, or more commonly, *LOP*s, in contrast to global operations like Boolean set operations. Typical representatives of this kind of modeler are all Romulus-kernel-based systems like HP PE/ME30.

The difference between modifications with Boolean operations and modifications with LOPs lies in the amount of context analysis required. A Boolean operation always works on the complete volume of the operands (global operation). A LOP only analyzes the neighborhood of the operand and is usually not able to perform topological changes. To perform a model modification several constraints have to be met by the model, two of which are illustrated in Figs. 1 and 2.

The example shown in Fig. 1 is a block with edge E1 to be blended (rounded). If the radius chosen for the blend is larger than the distance between the two edges E1 and E2, the topology of the model would need to be changed or the model would be corrupted.

Fig. 2 shows a block with a pocket on its left side. To move or copy the pocket from the left top face to the right one cannot be done using LOPs, because both top faces would change their topology (i.e., add or remove faces or edges). The left top face would lose the inner loop resulting from the pocket while the right top face would add one.

These two restrictions are only examples of the complex set of constraints on the use of LOPs. Removing these restrictions one by one means evaluating more and more scenarios, thus adding to the complexity of the algorithms needed for the operations. A new approach was necessary.

The Tool Body

The limitations illustrated above led to the question, why can't Boolean set operations do the job? Boolean operations would be able to handle all special cases and at the same time would increase the stability of the algorithms. In the late 1980s a lot of research and development was done using this approach. Two directions were taken. The first was to further develop the old-style CSG modeling systems to make them easier to use. The second was to remove the limitations of LOPs in systems like HP PE/ME30 and all other Romulus-kernel-based systems. HP took the latter approach to develop the dynamic modeling capabilities of HP PE/-SolidDesigner.

To enable model modifications with topology changes, Boolean operations were added to the LOP modification capabilities. The system generates a *tool body* and positions it according to the specifications of the modification. A Boolean operation between the original model and the tool body results in the desired model modification.

In this article, the term *basic local operations* (B-LOP) will be used for the normal LOP, which cannot perform topology changes, while the process of using the Boolean operation, if necessary or more appropriate, will be referred to as an *intelligent local operation* (I-LOP). Although the Boolean operation does not need to be done in all cases, the term I-LOP will be used to indicate that there can be a Boolean-based part of the operation.

To use the Boolean set operations for I-LOPs the system needs to create a tool body first. Two major approaches can be distinguished:

- Analysis of the the geometry to be modified and generation of an appropriate topological primitive (i.e., a basic volume element such as a cube, prism, or other) whose faces will be forced (tweaked) to build up the geometry of the tool body.
- Topological and geometrical creation of the tool body in only one step by analyzing the geometrical and topological neighborhood of the face to be moved.

The first approach is easier to implement if a utility function (a set of B-LOPs) is available that performs the task of tweaking a topologically similar object onto the required geometry of the tool body. The tweaking function, however, is tied to the restrictions of this utility function. The second method is more flexible but requires more knowledge about the internal structure of the CAD system's kernel.

We chose the first approach because HP PE/SolidDesigner already provided a working internal utility function for tweaking. The tool body generation for moving and tapering

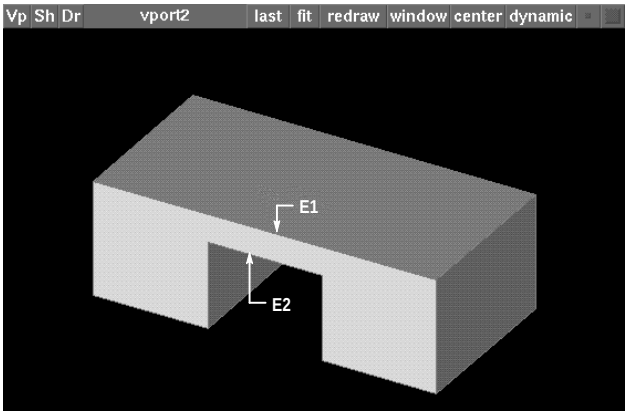


Fig. 1. An example of the limitations of local operations (LOPs). Edge 1 is to be blended (rounded). If the radius chosen is larger than the distance between E1 and E2, the topology of the model must be changed or the model will be corrupted.

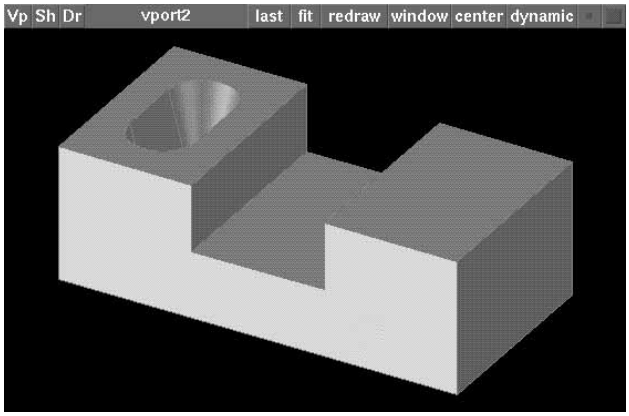


Fig. 2. Another example of the limitations of LOPs. The pocket in the left top face cannot be copied or moved to the right top face using LOPs because both top faces would change topology by adding or removing faces and edges.

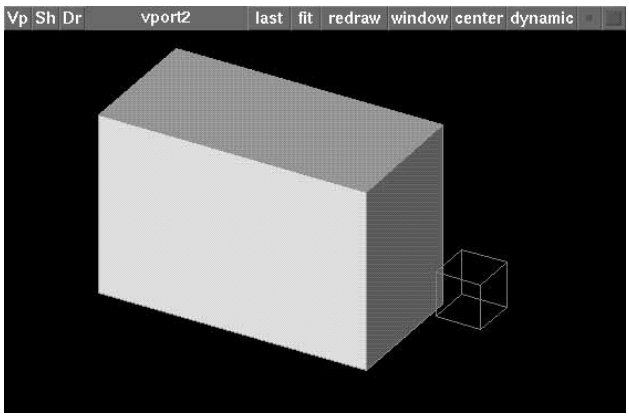


Fig. 3. The first step in the I-LOP (intelligent local operation) approach for a stretch (move face) operation in HP PE/SolidDesigner is the generation of the tool body, a four-sided prism in this case.

faces (and for bosses and pockets) follows two steps, which are carried out by the system automatically without any user intervention. First, a 3D body is created that has the topology of the final tool body. The part to be modified is analyzed to determine the topology of the 3D body that has to be generated for the requested operation. Depending on the number of edges in the peripheral loops of the face to be modified this body is either a cylinder (one edge), a half cylinder (two edges), or an n-sided prism, where n is the number of peripheral loops. Second, the geometry of this body is modified using basic local modifications. The result is the final tool body to be used for the model modification.

Figs. 3 to 5 illustrate this approach in further detail, showing the I-LOP approach for a stretch (move face) operation in HP PE/SolidDesigner. The user wants to stretch the box in Fig. 3, which means that the right face of the box will be moved to the right. The only and outer loop of the face to be moved contains four edges. Thus, the system creates a four-sided prism in space at an arbitrary position.

As shown in Fig. 4, the system then forces the faces of the prism onto the surfaces underneath the front, top, back, and bottom faces of the box (B-LOP). The left face of the prism will be forced onto the right face of the box and the right face of the prism will be forced into its final position, specified by the user.

The last step, shown in Fig. 5, is the Boolean set operation (in this case a unite) of the tool body with the original 3D part, resulting in the modified 3D part. Although the modification in this example could have been achieved by employing a B-LOP operation, the use of the Boolean set operation will allow topological changes like interference of the stretched 3D part with some other section of the model.

The same approach works for faces with outer loops of n-sided polygons. The curves describing the polygons are not restricted to straight lines. All types of curves bounding the face are valid, as long as the boundary of the face is convex. In cases of convex/concave edges special care has to be taken in tweaking the faces of the prism onto the geometry of the adjacent elements of the original part. An approach similar to the one described applies for tapering faces.

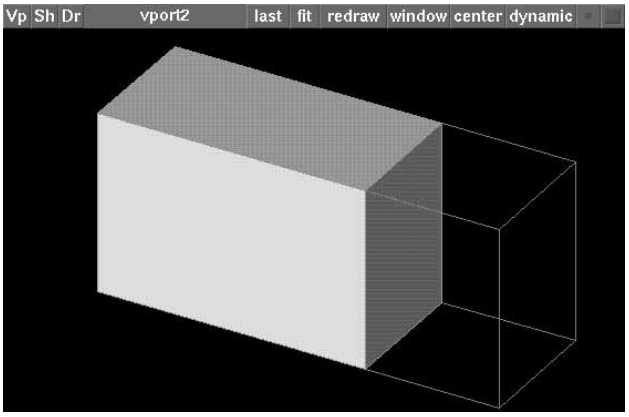


Fig. 4. The second step in stretching the box of Fig. 3 is to force the faces of the tool body to the final geometry, using a B-LOP (basic local operation).

There is a high risk of getting unpredictable results or self-intersecting tool bodies when dealing with several faces that are not related to each other. Although the example in Fig. 6 may look somewhat artificial, it is characteristic of many possible situations. The user wants to move the two vertical faces F1 and F2 farther to the right, and expects a result as represented by the right part in Fig. 6. However, depending on the sequence of selection, two different results can be obtained.

If F2 is selected before F1, the I-LOP performs as expected and the result is as shown at the right in Fig. 6. If F1 is selected first, however, F1 will be moved first. The tool body belonging to F2 will then be subtracted from the body and will interfere with the final position of F1. This leads to the unexpected result shown in the middle of Fig. 6.

The conclusion is that only single faces can be modified and change topology during the modification. For multiple faces the I-LOP is too risky. If multiple faces are to be modified at once, basic local operations (B-LOPs) instead of Boolean operations will be activated. No topology change is allowed, of course. One major exception to this rule is the case of bosses and pockets, which will be discussed later.

Although in most cases the I-LOP approach will be applied, there are situations where self-intersecting tool bodies would be created and therefore the B-LOP approach is preferred

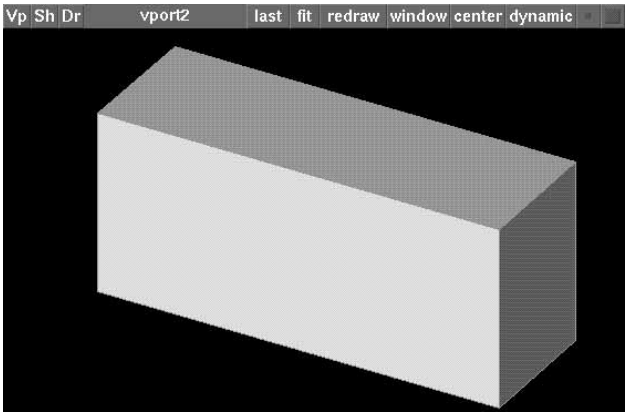


Fig. 5. The final step in stretching the box of Fig. 3 is to unite the tool body and the original part, using a Boolean operation.

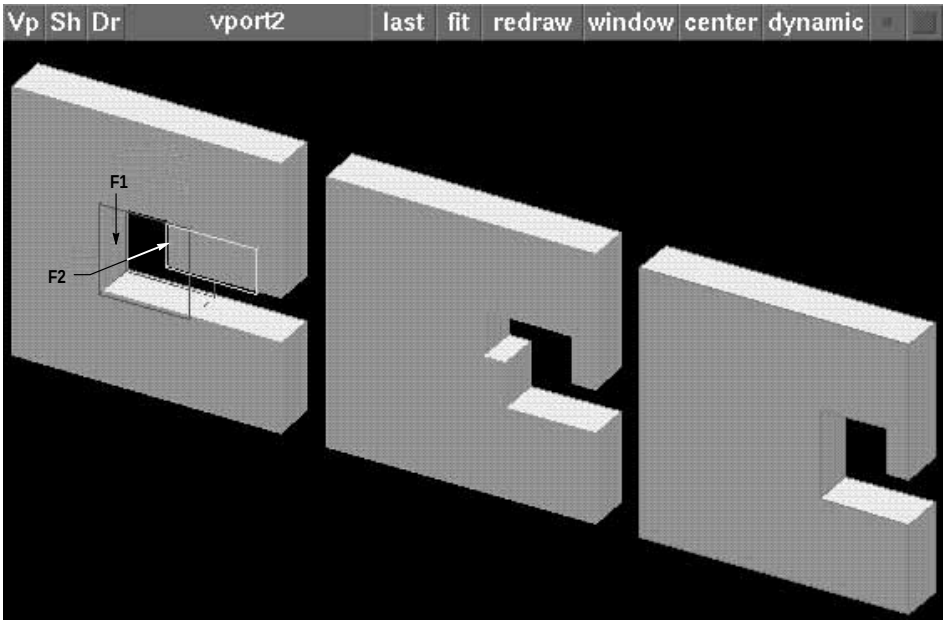


Fig. 6. Modification of several unrelated faces can lead to unanticipated results. Here the user wants to move faces F1 and F2 to change the part at the left into the part at the right. If F2 is selected before F1 the result is as expected, but if F1 is selected first the result is the part in the middle.

even in cases with only one face to be moved. Fig. 7 shows such a situation. The user wants to rotate the right face around an axis lying in the face itself. Another likely situation would be aligning the right face with another face of the model.

Using an I-LOP in the way described above, a self-intersecting tool body would be created without special care to dissect the tool body into two tool bodies, one to add to the part and one to subtract from the part. In Fig. 7, the volume to be added is colored green and the volume to be removed is red. If HP PE/SolidDesigner detects a situation like this, a B-LOP is used for the modification.

Geometry Selection and Automatic Feature Recognition

The next step in terms of increased complexity is the handling of groups of faces, which are known as bosses or pockets by mechanical engineers. These bosses and pockets need to be moved or copied, allowing topology changes. Of course the end user would appreciate it very much if these features could be selected as a unit as opposed to the cumbersome selection of faces sequentially.

First, the terms boss and pocket need to be further specified. Bosses and pockets can be defined as a number of connected faces whose exterior boundary loops (the edges describing the intersection of the tool body with the original 3D part) are internal loops of a face. This definition is not easily conceivable and can be replaced by the more understandable, yet not very exact definition, "a number of connected faces contained in one or two nonadjacent others." This is easily conceivable by the end user and fits a lot of cases. Figs. 8 and 9 illustrate the copying of a pocket to which this definition applies.

For moving or copying bosses or pockets the system dissects the part along the edges that connect the boss or pocket with the remaining part. Both the tool body (the former boss or pocket) and the part to be modified now have open volumes (missing faces, or "wounds"), which are "healed" by the algorithm before further processing with the tool body.

Figs. 8 and 9 show only simple pockets. The question remains of how to deal with more complicated situations like

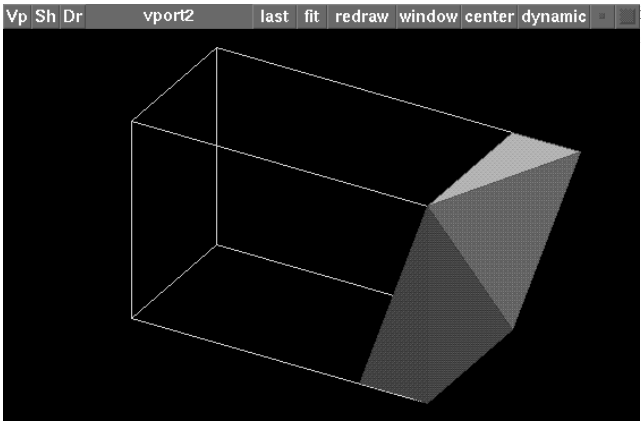


Fig. 7. Here the user wants to rotate the right face around an axis lying in the face itself. This would create a self-intersecting tool body if an I-LOP were used. HP PE/SolidDesigner detects such situations and uses a B-LOP instead.

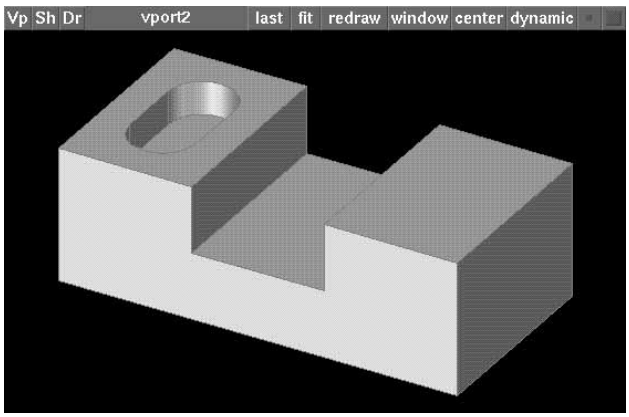


Fig. 8. A part with a pocket to be copied to the right top face.

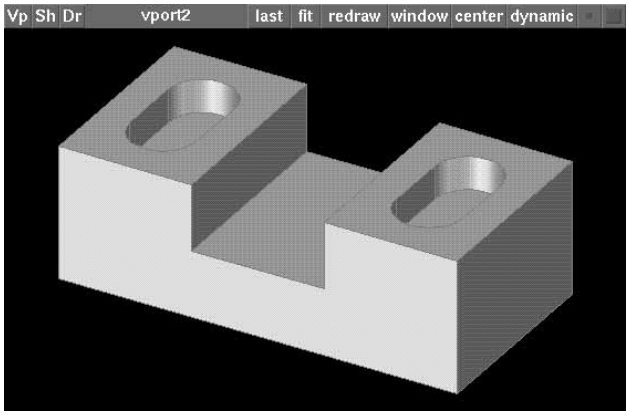


Fig. 9. The part of Fig. 8 with two pockets, one copied. The system recognizes simple and compound bosses and pockets.

countersunk holes or bosses inside pockets. Fig. 10 shows the extension of the simple bosses and pockets. A boss or pocket containing countersunk bosses or pockets will be referred to as a *compound boss or pocket*. Any number of nested bosses or pockets is allowed, as shown in Fig. 10.

Simple and compound bosses and pockets are recognized by the system automatically, depending on the selection of the user. If one face within the boss or pocket is selected, the feature recognition algorithm identifies all other faces belonging to the selected boss or pocket.

Fig. 11 shows a part with a countersunk pocket. If the user selects one of the red faces in Fig. 11, the whole pocket is selected. If the user selects one of the yellow faces a smaller pocket will be recognized.

Feature recognition very much simplifies geometry selection. Instead of many picks to sample the list of faces for a move or copy operation, one single pick is enough. HP PE/SolidDesigner recognizes the list of faces as a boss or pocket and the subsequent modification can include topological changes.

Once the bosses or pockets are selected, various I-LOPs are applied:

- The "wound" in the top face of the part to be modified is healed, resulting in a simple block and a tool body consisting of the two nested pockets (the colored faces).

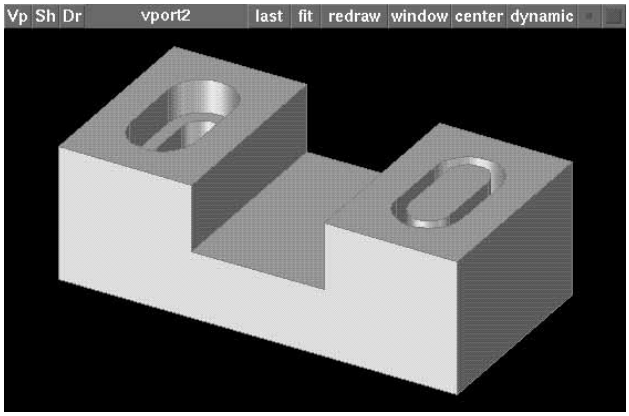


Fig. 10. Part with one compound pocket and a boss inside a pocket.

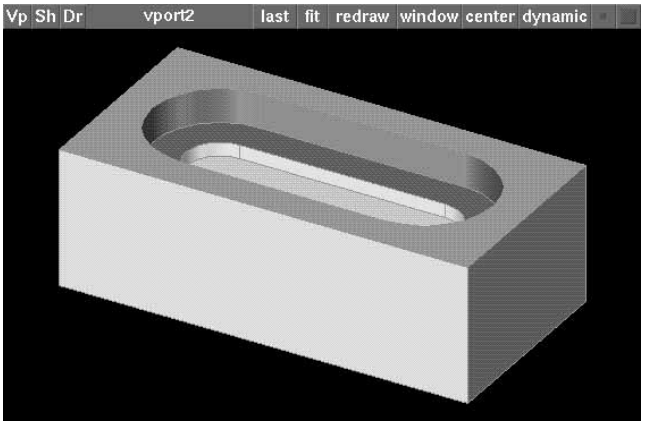


Fig. 11. Part with a countersunk pocket.

- This compound tool body is split into the larger pocket (colored red, nesting level 1) and a smaller pocket (yellow, level 2).
- Both tool bodies are transferred to their final positions.
- The larger tool body is subtracted from the block.
- The smaller tool body is subtracted from the result of the preceding, leading to the desired modification of the part.

The additional complexity of working with compound pockets or bosses is mainly handled by the Boolean engine of HP PE/SolidDesigner. Only a small part—the detection and subdivision of compound bosses or pockets—is needed in the I-LOP code itself.

Fig. 12 shows the result of tapering a compound pocket with HP PE/SolidDesigner. (The front corner of the block has been cut away to show the tapered pocket.) If there were a need to change the topology by this operation, the Boolean operation inside the I-LOP would take care of it.

These features in PE/SolidDesigner don't have anything to do with the generation method of the model, as is the case in history and feature-based modelers. The features are defined temporarily for specific purposes; they are not part of the model. The flexibility of defining features at any stage in the design process is very much appreciated by most mechanical engineers.

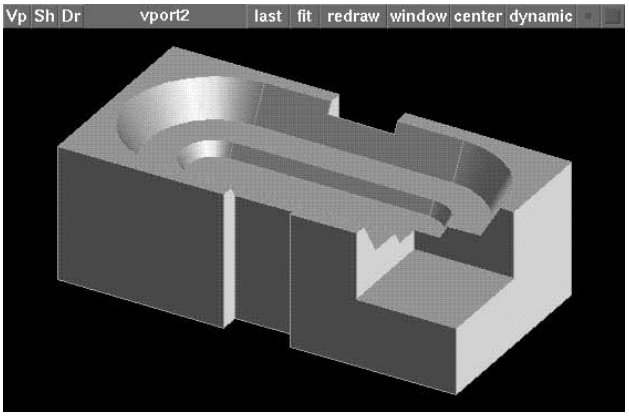


Fig. 12. Part with a tapered, countersunk pocket. The front corner of the block has been cut away to show the tapered pocket.

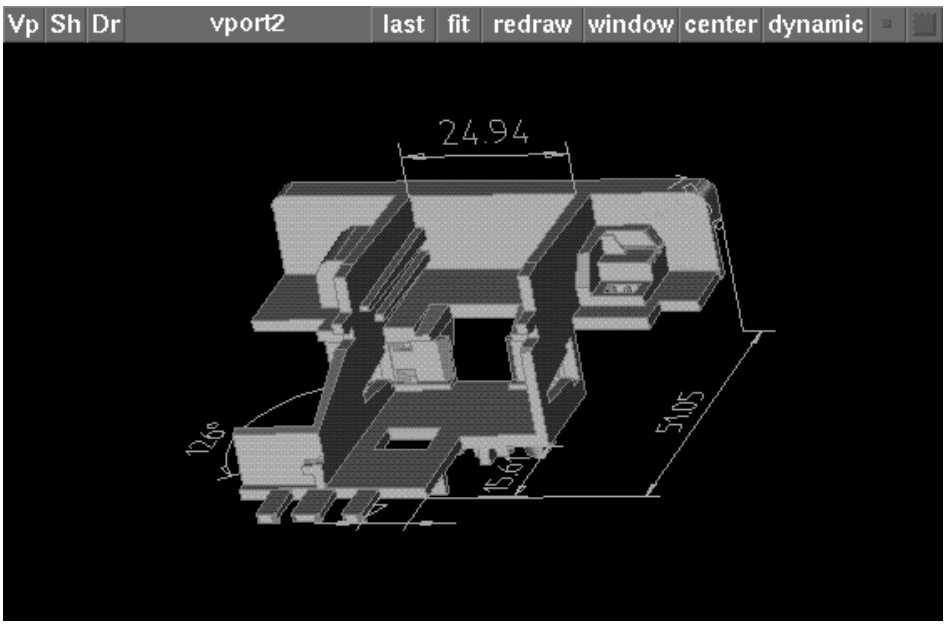


Fig. 13. Part of an HP DeskJet printer printhead.

3D Labels for Dimension-Driven Modifications

In the past, if a mechanical engineer or draftsman had to adapt an existing design to new dimensions, the easiest way was to mark the dimensions as "not true in scale," erase the original value and put in the new value. The rest was left to the people on the shop floor.

This concept of modifying labels was adapted by CAD systems that use variational or parametric approaches in either 2D or 3D. The difference between the parametric and variational approaches is minor in this respect. Both systems require a completely constrained drawing or 3D model which is generated with the help of user constraints and system assumptions. New values of the dimensions cause a recomputation of the whole model. Any dependencies that the user might have specified are maintained even when the model becomes modified later in the design process. The design intent is captured in the model. While this approach is most efficient for family-of-parts designs, it does not support flexible modifications, which are needed in the typical iterative design process.

HP PE/SolidDesigner's dynamic modeling capabilities support the concept of 3D labels that can be attached to the model at any time during the design process and can be used as *driving values*. Tapering of the selected geometry can be driven by angled labels, while the transformation of the selected geometry can be defined by employing distance labels. The user adds one or several 3D labels to the part, selects the geometry to be modified, and specifies new dimension values. Using the new values the system then performs the modification employing B-LOPs or I-LOPs. After the modification all values of the labels are updated to the current values of the geometry.

Fig. 13 shows the HP PE/SolidDesigner model of a part of the printhead of an HP DeskJet printer. Figs. 14 through 18 illustrate the concept of 3D labels.

As indicated in Fig. 14, the first draft of the design contained a 30-degree ramp that was to be used to aid manufacturing. All edges of the area are blended to meet casting requirements. Assume that later in the design process it turned out that the ramp was not needed at all or a different angle was needed. There are several ways to define the transformation in space for the ramp to disappear (e.g., aligning the original ramp face and the adjacent face below the ramp). If the user is trying to define the axis of rotation for the ramp face,

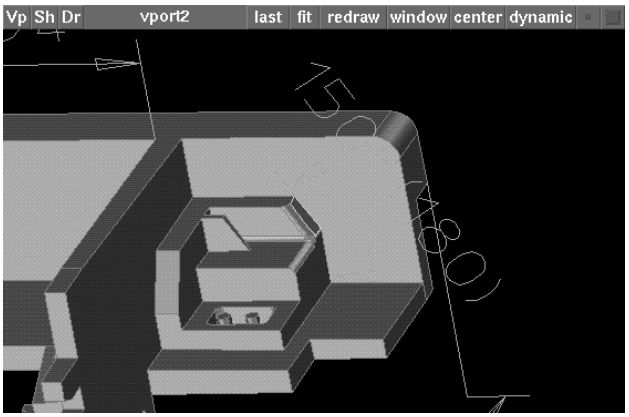


Fig. 14. Changing a dimension (the angle of the ramp) of the part of Fig. 13.

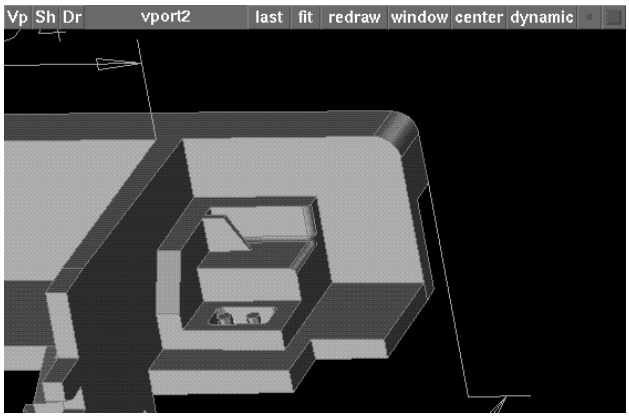


Fig. 15. The part of Fig. 13 with the new ramp angle (the ramp has been removed).

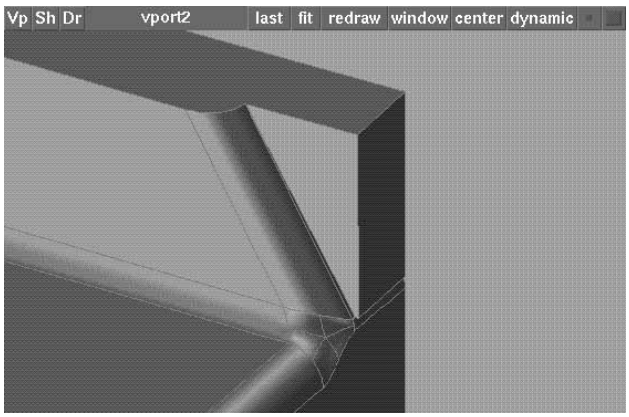


Fig. 16. The part of Fig. 13 changed by I-LOPs without the knowledge that there is a blended edge.

problems arise because the axis is a virtual one and cannot be found in the model. Either a special method for axis definition is needed or the user has to do the calculation by hand.

A third possibility is employing 3D labels. Using the 3D label already defined to show the functional angle enables the system to do all the necessary computation. A new value (in this example 180 degrees) needs to be entered by the user. The system derives the transformation that has to be applied to the ramp face and the model becomes updated. (Fig. 15).

If the label had not been ready for use, it could have been created to drive the modification. The labels are independent from the model creation and can be used temporarily. If the model has been changed, the values of the dimensions update automatically to their new values.

Relation Solver

Once the geometry to be modified is selected and new values of the labels are entered, the system will start with the unspecified transformation and six degrees of freedom (three translational and three rotational). The solver will derive the relations from the labels and reduce the number of degrees of freedom sequentially one after the other until all specified relationships are satisfied or an impossible configuration is encountered.

The system is only designed to solve relationships that can be described by equations solvable by algebraic means. No iterative solution is attempted.

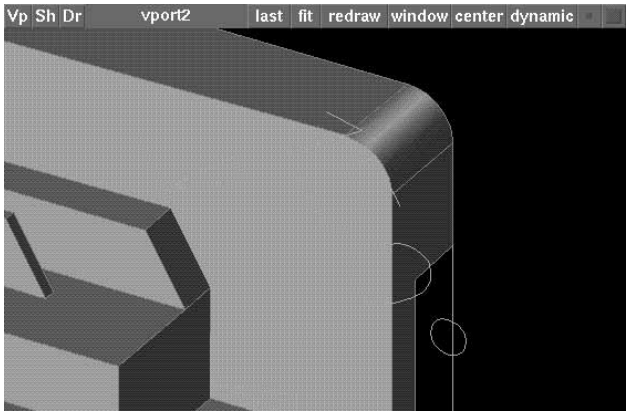


Fig. 17. HP PE/SolidDesigner avoids the behavior of Fig. 16 by first suppressing the blend as shown here.

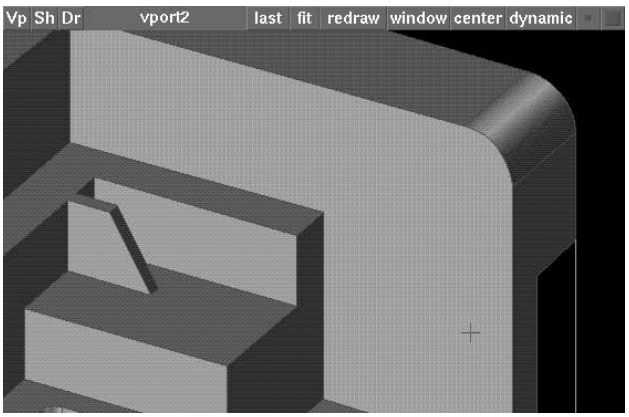


Fig. 18. After suppressing the blend, the system makes the change as shown here. The final step is to readd the blend as shown in Fig. 15.

The resulting transformation is dependent on the order in which the user has selected the modification-driving labels. Thus, the result of the modification is order dependent, especially if rotational and translational transformations are specified for the same modification.

Modifying Blended Faces

In Fig. 14, there are blends adjacent to the face to be moved. If the system didn't know that there were blends in the neighborhood of this face and how to handle them, moving the face might create a strange object like the one shown in Fig. 16.

To avoid this behavior, the system suppresses the blends in a preprocessing step before doing the main operation (rotate the ramp face) and recreates them after performing the main operation in a postprocessing step. Figs. 17 and 18 show the steps used by the system internally.

This concept adds to the flexibility of HP PE/SolidDesigner tremendously, because it overcomes the limitation of the B-LOPs that only modifications can be done that do not involve topological changes.

Summary

This paper shows the strengths of the dynamic modeling techniques. Topology changes are possible in most cases. Model modifications can be defined when they become required within the design process. Design changes do not have to be anticipated when starting the model creation. No constraints within the model exist, and predictable results avoid the trial-and-error approach of parametric and history-based systems. Dynamic modeling's core component besides the relation solver is the tool body, which is defined by the system automatically for the Boolean operation during a model modification. Although some limitations exist, most design changes are possible in one or several steps.

Acknowledgments

We would like to thank all those who helped with the development of dynamic modeling, in particular the Mechanical Design Division HP PE/SolidDesigner team, which was supported by 3D/Eye of Ithaca, New York and D-cubed of Cambridge, UK.

User Interaction in HP PE/SolidDesigner

The HP PE/SolidDesigner user interface is modeled after the successful, easy-to-use, easy-to-learn interface of earlier HP CAD products. All commands are coded as Common Lisp action routines. A user interface builder helps command programmers by hiding details of the X Window System and the OSF/Motif™ graphical user interface. Prototyping was done using a specially developed Lisp-based interface to OSF/Motif called HCLX.

by Berthold Hug, Gerhard J.Walz, and Markus Kühl

As the use of CAD systems has become more and more widespread, two conflicting trends have emerged. On one hand, the complexity of CAD systems has grown with their increasing functionality. On the other hand, the typical CAD system user is no longer a computer hobbyist. Designers and detailers are busy enough maintaining expertise in their own areas without having to be computer experts as well. Therefore, CAD software must be easy to learn and easy to use for first-time or occasional users without sacrificing flexibility and effectiveness for more experienced users. The conflict between the need for simple operation and the increasing functional complexity can lead not only to less user satisfaction, but also to decreased productivity. As a result, a simple and consistent user interface has been a long-standing goal of HP CAD products.

The user interface of HP PE/SolidDesigner is based on the successful user interface of HP PE/ME10 and PE/ME30. The key components of this user interface are:

- Ease of Use. The product is designed not only for experts, but also for first-time or occasional users.
- Menu Structure. A task-oriented, flat menu structure minimizes menu interaction and the length of cursor movements.
- Macro Language. This allows the user to customize the menu structure. User-defined functions can be set up to increase productivity by using existing CAD operations and measure/inquire tools for model interaction.
- Online Help System. This provides all relevant information to the user without using manuals.

The HP PE/SolidDesigner graphical user interface is based on OSF/Motif and the X Window System, universally accepted graphical user interface standards for applications software running on workstation computers. The OSF/Motif graphical user interface provides standards and tools to ensure consistency in appearance and behavior.

The large functionality built into HP PE/SolidDesigner is accessed by means of a command language with a defined syntax, referred to as *action routines*. The user communicates with the command language via the graphical user interface. All prompting, error checking, and graphical feedback are controlled by means of the command language. All CAD functionality is provided in this way, along with a user interface builder for implementing the graphical user interface.

The action routines are implemented in Common Lisp, which provides an easy and effective way of prototyping and implementing user interactions. For the first interactive prototypes, HCLX, a Lisp-based OSF/Motif interface, was used.

During the development of HP PE/SolidDesigner, HP mechanical engineers spend hundreds of days testing the product and providing feedback to tune its user interaction to meet their needs. They mercilessly complained about any awkward interactions. They made suggestions and drew pictures of how they would optimize the system for their particular tasks. As a result, commands were designed and redesigned to reflect their needs. The user interface verification was also supported by many external customer visits.

User Interface Description

If the user is familiar with other OSF/Motif-based applications, it's easy to feel comfortable with HP PE/SolidDesigner quickly. The mouse, the keyboard, and the knob box or spaceball are the tools for interaction.

When HP PE/SolidDesigner is started it looks like Fig. 1. The different areas are:

- Viewport (center of the screen). The viewport covers the main portion of the user interface and consists of the graphics area and the viewport control buttons at the top. In the graphics area of the viewport, the model is displayed and the user interacts with the model. Several viewports can exist, each with its own control buttons. Using more than one viewport, the user can view a part simultaneously from different sides and in different modes. Resizing and iconification of viewports are possible.
- Utility Area (top row). In the utility area, the user finds utility tools that support the current task. They do not terminate, but rather interrupt and support the current command. The help button at the right end gives access to the general help menu.

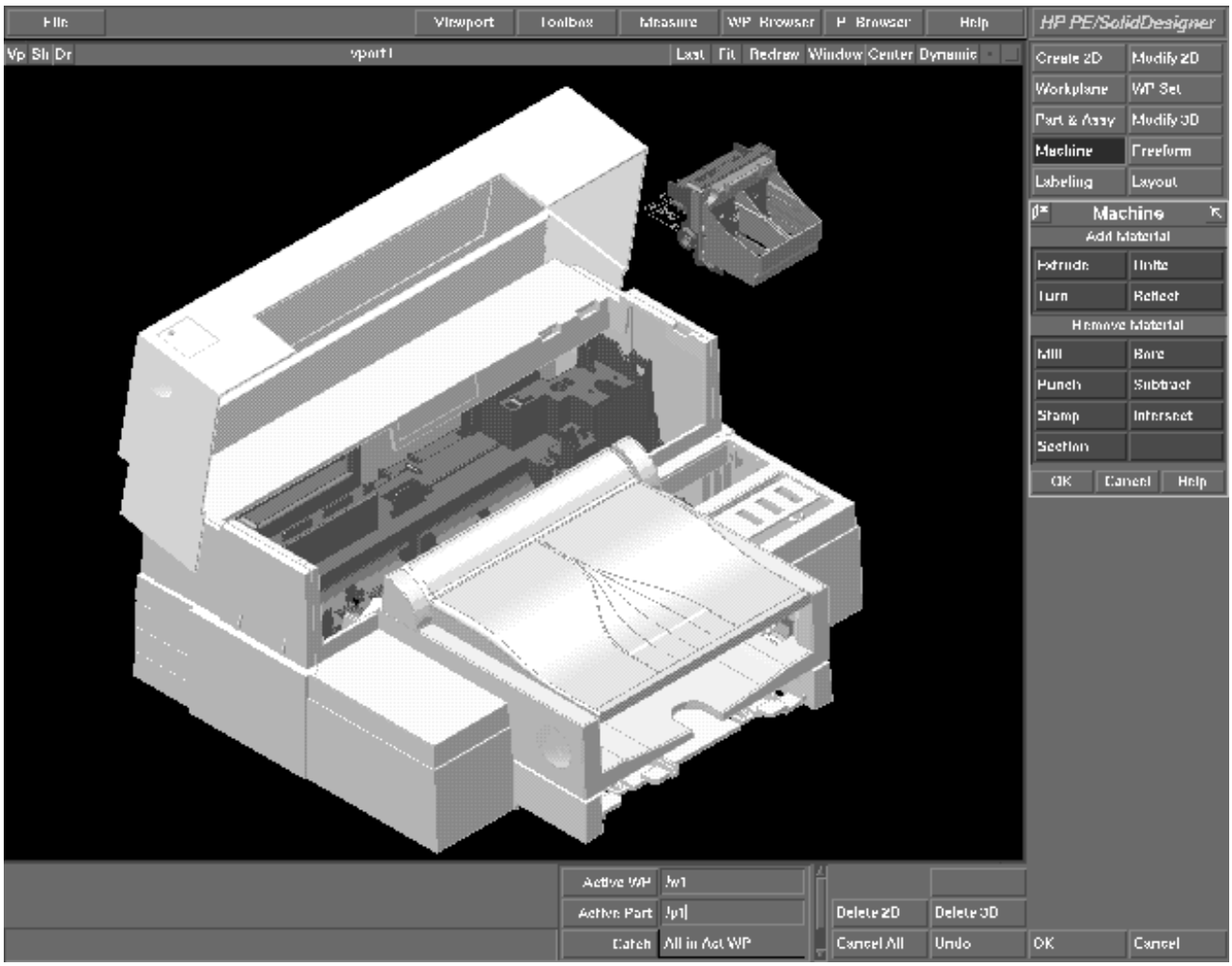


Fig. 1. Main screen of the HP PE/Solid Designer user interface.

- Main Menu (right side). The main menu buttons appear in the right column below the application name. This is also called the main task area. All the functionality is grouped into task-oriented logical areas. By selecting a main task button, the user opens a set of subtasks or a command dialog menu with buttons for all stages in the modeling sequence.
- Prompt Lines and General Entry Field (bottom left). The two-line prompt area is used for general system feedback, messages, or user guidance. The general entry field is used for entering commands, general expressions, and the like.
- Global Control Buttons (bottom right). The buttons at the bottom are always available. The select button is only active when the system is prompting the user to select something. The buttons and display fields inside the scrolled windows display general system settings like the active workplane or part, units, and catch information.*The other buttons are commands that the user needs frequently. They are always available.

*Depending on the current command, the catch setting indicates how a pick in the graphics area (viewport) is processed to identify an element. For example, "catch vertex on current workplane" means that if the user picks near the end of a straight line, the resulting pick point will exactly match the endpoint of the line. The catch radius is customizable.

Command and Option Dialogs

Command dialog boxes (see Fig. 2) are accessed either from the main task area or the utility area. The current command dialog box is replaced by the new selected one. If the default home position of the command dialog box is inside the drawing area, the dialog box is closed upon completion of the operation (this is typical for command dialogs from the utility area). With this behavior the user always has optimal use of the screen space.

Nevertheless, sometimes the user wants to have parallel access to different dialog menus at the same time (flat structure). This can be achieved by pinning the command dialog to the screen using the small icon in the upper left corner. Pinned command dialog boxes are helpful whenever the user is using several menus constantly. The user can keep as many or as few dialog boxes open as desired and arrange them on the screen to suit the present task. Fig. 2 shows two pinned dialog boxes and one unpinned dialog box.

Activation of a command by a mouse click or by typing in a command in the general entry field leads to the same behavior. The command button snaps into pressed mode. If there exist a number of additional controls of the command, a

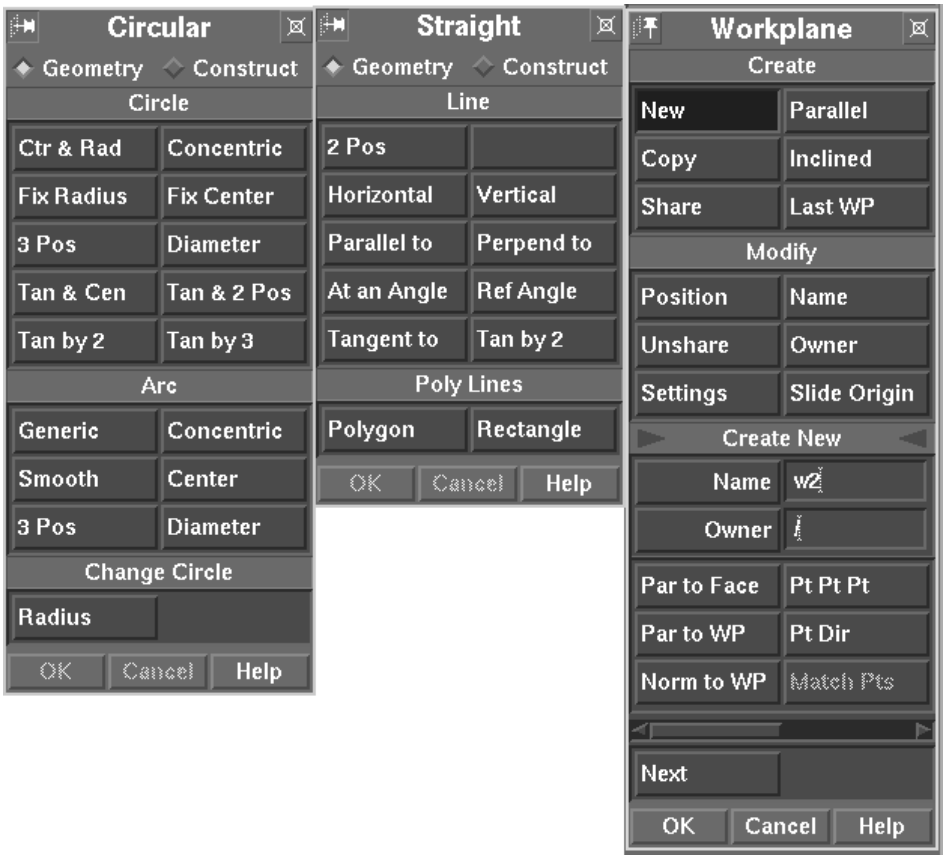


Fig. 2. Command dialog boxes with pin icons in the upper left corner. Two boxes are pinned to the screen and one is not.

subdialog is attached at the bottom of the command dialog box (see extrude box in Fig. 3). The command becomes interactive and a prompt asks for further input. The dialog box gets a yellow border, a signal that this dialog box is active. If the action is suspended by an interrupt action, the border changes to red. Thus, the user never loses track of what is active and what is not.

The subdialog provides options in the form of buttons, data entry fields, and check boxes for further control of the command. The system provides good defaults to minimize the required user input. All options can be manipulated in any appropriate order; the command supplies a parallel syntax. All settings are displayed in the dialog box. Required data fields are highlighted in yellow, meaning that the user must define a value.

The help buttons of the command dialog boxes give access to context-sensitive help.

Context-Sensitive Help

Help messages relating directly to the task the user is performing can be accessed immediately by pressing the help button located in the currently active menu or dialog box. The help information appears in its own dialog box, which can be positioned anywhere on the screen and resized for convenience (see Fig. 4).

Words used in help text are directly linked to other definitions or explanations. The user need not go back to indexes to look up further words to aid in understanding the help information.

Fig. 3. If a command has controls in addition to the basic ones, a subdialog box is attached to the command dialog box. The extrude command is an example of this behavior.

In addition to the context-sensitive help, the help system provides a task-based index with search facility, a command-based index with search facility, an overview of HP PE/SolidDesigner, information on HP PE/SolidDesigner's concepts, filters, and displays of user-typed keywords, version information, and help on help. The help system can be used in a standalone mode without running HP PE/SolidDesigner.

Task-Sensitive Tools and Feedback

Whenever the user has to enter a value for a command, the system provides the appropriate tool for data entry. For instance, if the user has to enter a direction, the direction tool (Fig. 5) pops up. The user can extract the information directly out of the model with a minimum of effort by accessing parts of the model such as edges and faces. The result is displayed either textually or graphically as part of the model.

These task-sensitive tools are implemented as subactions so that all commands (action routines) have access to the same tools. Using these tools guarantees consistent system behavior, for example in specifying directions.

Browsers

Browsers (see Fig. 6) display lists of files, workplanes, parts, and assemblies, and allow selection of items for use in commands without typing in names. Even complex assemblies become easy to understand and manipulate when browsers are used.

Customizing the User Interface

HP PE/SolidDesigner provides different facilities for changing its user interface. The following customization capabilities exist:

- Flattening the Menu Structure. This facility is provided by allowing the user to pin command boxes to the screen. When the environment is saved, pinning and location information is stored for later access.
- Toolbox. The toolbox (Fig. 7) allows the user to build a custom command dialog box. The user can put any command into the toolbox, and can put the most-used commands together in one area for easy access. The toolbox can be left open like a command dialog box. If a command becomes interactive, the original subdialogs are attached at the bottom of the toolbox dialog.
- Lisp. The user can write Lisp functions, which can contain action routine calls. Thus, the user can combine Lisp with CAD functionality to optimize the system for particular needs.
- Key Button Bindings. HP PE/SolidDesigner commands or Lisp functions can be accessed via X translations. Function keys, mouse buttons, or any key sequence can be defined for accessing any given functionality. This tool allows the expert user to accelerate the use of the system.
- Record/Playback. The record/playback feature allows the user to record a series of command picks to be used later to duplicate the action, like a macro. The information is stored in a file for playback. The file contains the command syntax, so it can be used to support writing user-defined Lisp functions.

Action Routines and Personality

This section describes the user interaction in HP PE/SolidDesigner in more detail. It explains the basic technology underlying the concepts that were described in the preceding section. A simplified extrude example is used to clarify the explanation.

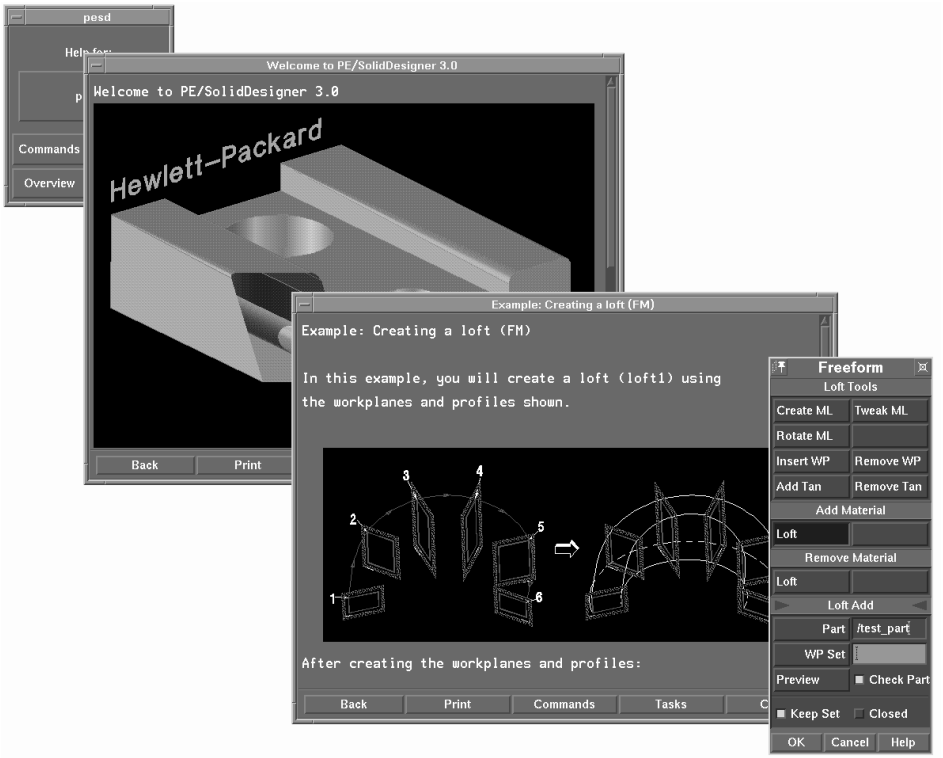


Fig. 4. Context-sensitive help information appears in its own dialog box, which can be positioned anywhere on the screen and resized for convenience.

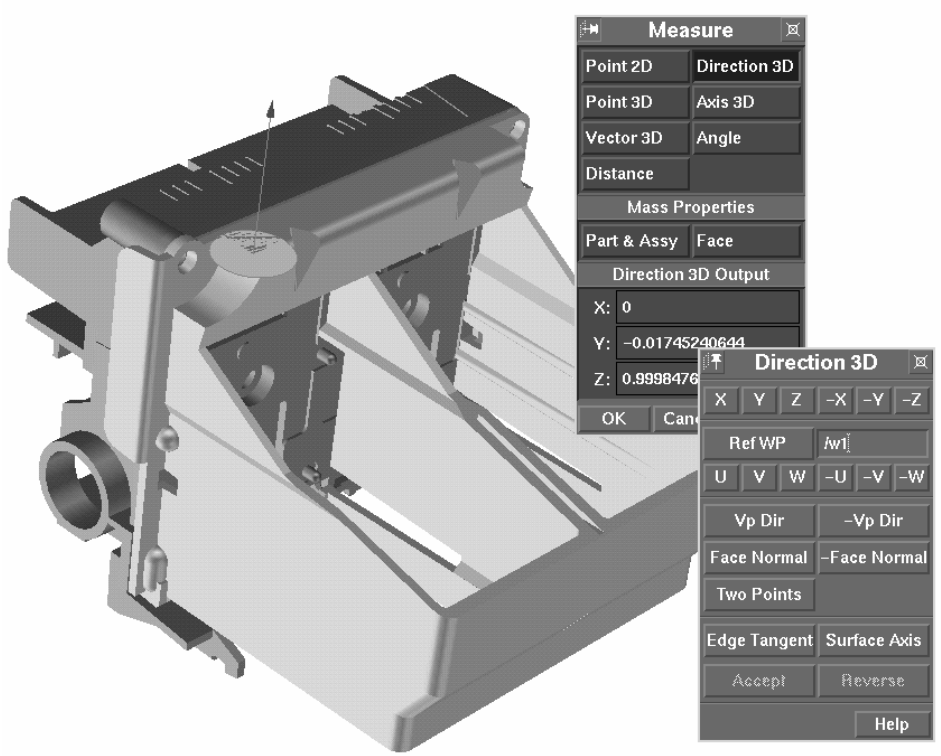


Fig. 5. When the user has to enter a value for a command, the system provides the appropriate tools for data entry. The result is displayed either textually or graphically as part of the model.

Fig. 8 is a simplified diagram of the action routine/personality communication model of HP PE/SolidDesigner. The communication model is divided into three parts. On the left side are the action routines and on the right side are the user interface objects. Bidirectional communication between the action routines and the user interface is the task of the *personality*, which is shown in the middle of Fig. 8. This division into three separate components allows the implementor of an HP PE/SolidDesigner command to change the user interface and its behavior without changing the command syntax. It is also possible to switch off the user interface for certain commands.

The action routine concept is used to implement the command language of HP PE/SolidDesigner. A command is coded as a state machine with several states and transitions between these states. The term personality refers to the information coded in the GUI update table shown in Fig. 8.

HP PE/SolidDesigner distinguishes three types of action routines:

- Terminate Actions. Terminate actions terminate every other running action routine *negatively* (i.e., they cancel them). At any time there can only be one *active* or *suspended* terminate action. All action routines that modify the solid model must be defined as terminate actions.
- Interrupt Actions. Interrupt actions interrupt the current running action routine. When the interrupt action is finished, the interrupted (suspended) action routine continues from where it was interrupted. There is no limit on the stacking of interrupt actions. Interrupt actions must not modify the solid model. They are only allowed to inquire about model data. A measure command is an example of an interrupt action.

Fig. 6. Browsers make complex assemblies easy to understand and manipulate.

Fig. 7. The toolbox allows the user to build a custom command dialog box containing often-used commands.

- Subactions. Subactions are used to implement frequently used menus so that they can be reused in other action routines. This avoids code duplication, allows better maintenance, and improves usability. Subactions can only be called from within other action routines. This means that the user cannot call a subaction directly. Some typical examples of subactions are:
  - Select
  - Measure axis, direction, point
  - Color editor
  - Part positioning.

Basic Action Routine Structure

As mentioned above, the user interface in HP PE/SolidDesigner is Lisp-based. Therefore, the implementation of an HP PE/SolidDesigner command using the action routine concept is a kind of Lisp programming. The following is a schematic representation of a terminate action:

```
(defaction name

  ( ) ;; List of local variables (with or without initialization)

  ( ;; action description

  ( statename (state_form)
                        (state_prompt)
                        help-index-symbol
     ( transitionpattern   (transition_form)   pers-update-symbol   next_state
)

     ... ;; more transitions

   )

   ... ;; more states

  ) ;; end of action description

   ( ;; local functions

   (local-fun ()

     ...

   )

   ... ;; more local functions

  ) ;; end of local function definitions

)
```

The structure of an interrupt action or subaction is equivalent to that of the terminate action shown above except that an interrupt action is defined using the keyword defiaction and a subaction is defined using the keyword defsaction. The second parameter of the action routine definition is the

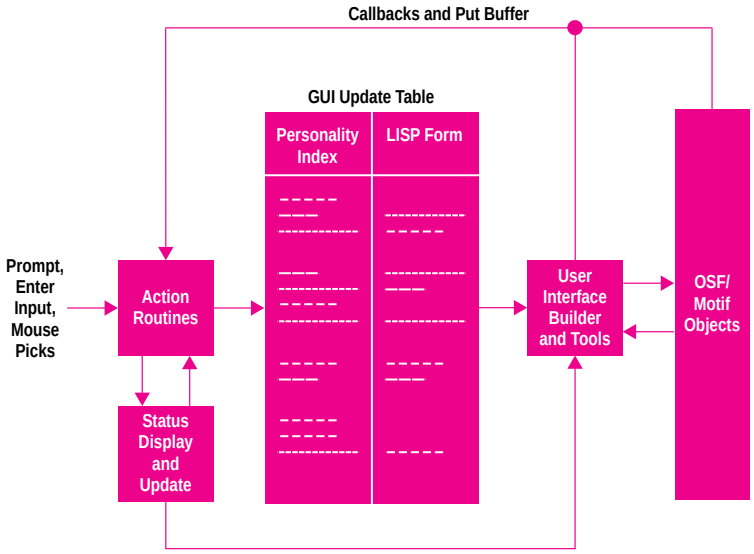


Fig. 8. In the HP PE/SolidDesigner user interface communication model, the action routines representing the commands communicate with the user interface objects through the personality.

name of the command that is coded through the action routine. For an extrude command this would be extrude. Following the command name is a list of local action variables. These variables can only be accessed from within this action routine. Action routine local functions and each state and transition form have access to them. They are used to store user-entered command parameters and as variables to control the execution of the command.

Next comes a description of the state machine. The states are those defined by the *railroad* of the command plus internal administrative states. The railroad of a command is a structure used to describe the syntax of an HP PE/SolidDesigner command for the user. Fig. 9 shows the simplified railroad of the extrude command (a few options have been omitted for clarity). The railroad reflects the concept of parallel command syntax. Each keyword (:part, :wp, :distance) can be given at any time and the command loops until the user completes or cancels it.

A distinction is made between prompting and nonprompting states. A prompting or prompt state requires the input of a token (a keyword or parameter value) from the user. This token is read from the input stream, which is filled either interactively by the user (hitting an option button, entering a number, selecting a part, etc.) or from a file (such as the recorder file). As many tokens as desired can be entered into the input buffer. Entered tokens are processed by the action routine handler. Processing stops as soon as an error occurs (such as an unknown keyword) or the input buffer becomes empty. HP PE/SolidDesigner then becomes interactive and requires more input from the user. A prompt state with an empty input buffer displays the prompt coded in its state.

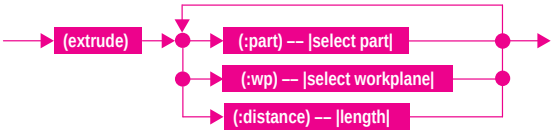


Fig. 9. Simplified railroad giving the high-level syntax of the extrude command.

After the user has entered a token, the action routine handler tries to match the input with one of the state transitions. If a match is found the action routine handler processes this transition and jumps to the next state. A nonprompting state (administrative state) takes the result of its state form to find a match with the coded transitions of this state. If the action routine handler was not able to find a match in the transitions and no "otherwise" transition was coded, it signals an invalid input error.

Implementation of the extrude railroad leads to the state machine shown in Fig. 10. As the extrude command starts, the first state is init. In this state the local variables are initialized and filled with useful defaults such as the current part and the current workplane with a valid profile. Since init is a nonprompting state and only one "otherwise" transition is coded the action routine handler goes on to the next state, top-prompt. This prompt state and the nonprompting dispatch state top-opt are the central states of this example command. The top-opt state takes the input of the previous state (top-prompt or any extract or check state) and tries to match its transitions. The states select-part and select-wp call on their only "otherwise" transition, the select subaction, as their transition form, with the specific select focus of part or

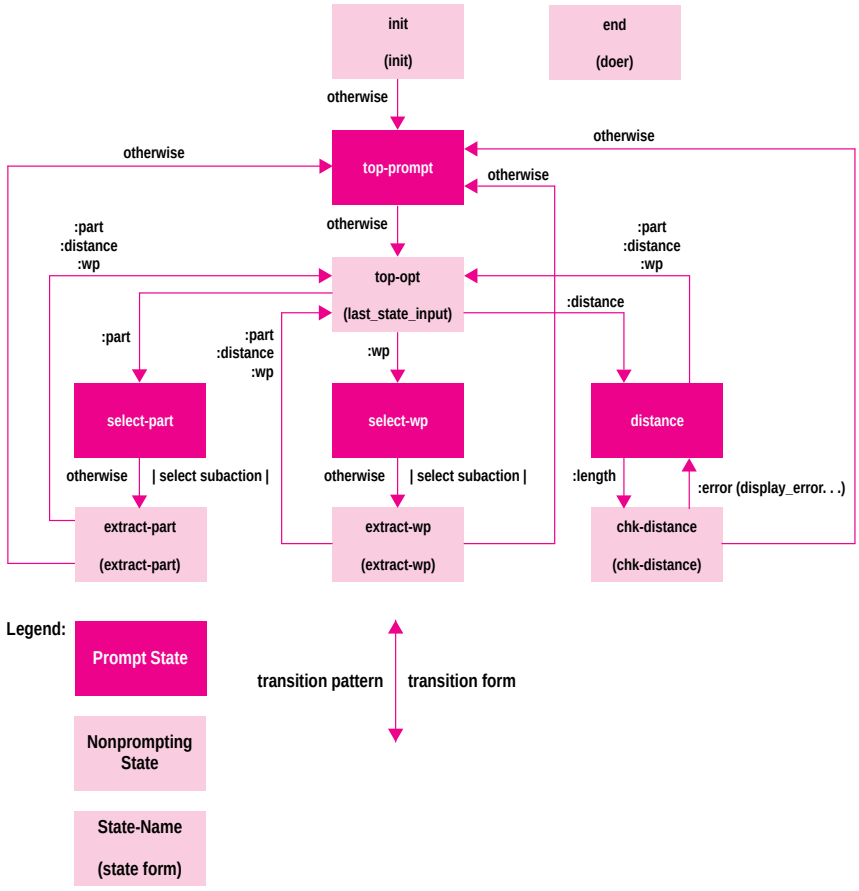


Fig. 10. State machine for the extrude command.

workplane, respectively. These states prompt through the select subaction. The extract states take the result of the transition form (select subaction call) and process the result of the select operation. The distance state has a special keyword—:length—as its transition pattern. For this keyword an input conversion is involved. The transition pattern will match any entered number, whereupon a units converter will be called automatically. A user can work in length units of millimeters or inches, and the units converter converts the length into the internal units (here mm). There are also other converters such as the angle converter which converts the user input (e.g., degrees) into internal units (here radians).

The extrude command loops until the user completes or cancels the command. In both cases the action routine handler jumps into the separated state end. Depending on a positive (complete) or negative (cancel) termination of the command, the software that actually performs the action will be called with the parameters that were collected by the action routine.

Personality

As explained earlier, the task of the personality is bidirectional communication between the action routine and the user interface objects. The core of the personality is the GUI update table shown in Fig. 8. This table stores all of the actions to be performed when an action routine executes, and it also receives data from the user. It guarantees that the user interface is in sync with the action routine state whenever HP PE/SolidDesigner requires data from the user.

The GUI update table is realized as a hash table with the pers-update-symbol (see action routine representation, page 19) as key and a Lisp form as entry. As soon as the action routine handler finds a match in the transition pattern of the current state it performs the transition form and triggers the user interface update using the third parameter of the transition definition as value. The action routine handler looks up whether a Lisp form is coded for the pers-update-symbol and evaluates it if found. The Lisp form can contain things like set-toggle of a command option or update-toggle-data to show the value the user has entered. This mechanism reflects the state of the action routine and its values at any time in the user interface.

There are special personality keywords for every action routine:

- ' action_name_ENTRY
- ' action_name_EXIT
- '(action_name action-interrupt-by-iaction)
- '(action_name action-continue-from-iaction).

' action_name_ENTRY is triggered as soon as the action routine starts. Normally the Lisp form coded for this entry ensures the display of the command options filled with all default values. ' action_name_EXIT cleans up the user interface for this command and removes the options from the screen. The other two keys are triggered when the command is interrupted or when it resumes its work after an interrupt action. In this case the coded Lisp form normally deactivates and reactivates the command options, since they are not valid for the interrupt action.

Delayed Update. A sequence of action routine calls (e.g., from the recorder file) or the input of several tokens into the input buffer should not cause constant updating of the user interface. Delayed update means that the user interface will not keep track of the action routine until the action routine becomes interactive, that is, until it requires data input from the user. At that time the user interface of the interactive command will reflect its state and values exactly.

A completely parameterized action routine does not cause any reaction on the user interface. If a command changes any status information (e.g., current part), this information will be updated. These updates bypass the GUI update table using the event mechanism.

The delayed update mechanism is implemented using a *personality entry stack*. Each trigger of a pers-update-symbol through the action routine handler will not lead to a direct execution of the Lisp form. All triggers are kept on the personality entry stack until the action routine becomes interactive. If an action routine doesn't require data from the user, all entries between and including ' action_name_ENTRY and ' action_name_EXIT are removed from the stack. As an action routine becomes interactive all Lisp forms belonging to the personality entries on the stack are performed until the stack is empty. The user interface is again in sync with the action routine state.

A problem came up with fully parameterized action routines behind a command toggle. Normally the ' action_name_EXIT trigger cleans up the command user interface, but with a fully parameterized action routine no personality trigger occurs. To solve this problem the system triggers two additional personality entries which are called in either delayed or undelayed update mode. These are ' action_name_PRE_ENTRY and ' action_name_POST_EXIT. The release of the command toggle is coded in ' action_name_POST_EXIT. The need for ' action_name_PRE_ENTRY is discussed below.

Personality Context. One requirement for the user interface of HP PE/SolidDesigner was that a command should be callable from other locations as well as from the default location. The motivation was the toolbox, which can be filled by the user with often-used commands. The main requirement was that a command's behavior in another context should be equivalent to its behavior in the default context. A user who calls the extrude command out of the toolbox expects the extrude options in the toolbox and not those in the default menu. The toolbox concept is based on the assumption that a command context is specified by:

- A calling button
- A dialog shell, in which the calling button resides
- A communication form where the command options are shown
- A shell position where the command options are shown if they are realized in a separate dialog shell.

All other things are command-specific and independent of the context.

The default context of a command is coded in ' action_name_PRE_ENTRY. Here the programmer of the command's personality defines the context in which the command should awake as the user types it in. This context can be overridden when the command is called out of, for example, the toolbox. Context dependent calls of the command personality have to check the current context settings

instead of having this behavior hardcoded in the default context. This concept also makes it possible to program a totally different personality for a command or to switch off the user interface of a command.

Stacked Personality. The possibility of invoking the same interrupt action several times makes it necessary to provide a method of creating independent incarnations of the interrupt action user interface. This is done by separating the building instructions of the command option user interface into a Lisp function. As an interrupt action is called a second time (or third, etc.) after an initial invocation, the widgets of the latest command option block are renamed to save the state and contents. Then a new incarnation of the option block is created using the building instruction function. When the most recent interrupt action terminates its execution the user interface incarnation is destroyed and the widgets of the saved option block are renamed again to become valid once more. One incarnation of the option menu of a command is always kept. All other necessary incarnations are created and destroyed at run time.

User Interface Development Tools

To speed up the user interface development process a prototyping tool was required that would allow modifications to be made quickly. Since the command language of HP PE/SolidDesigner is Lisp-based and the commands are intended to interact closely with the graphical user interface (GUI), standard C/C++-based user interface builders could not be used as prototyping tools. Such tools would have required the standard edit/compile/link/test cycle, which slows down the prototyping process heavily. They also didn't offer Lisp interfaces or facilities to change the GUI of the CAD system at run time, a required feature.

In 1989 only a few Lisp interfaces to the X and OSF/Motif toolkits were available. Because none of these had all of the features we needed, we decided to produce our own. Called HCLX, it is a Common Lisp interface to the X11 Xlib, the X toolkit intrinsics, and OSF/Motif widgets (Fig. 11). It provides Lisp functions for all the functions available in libX11, libXt and libXm, as well as all the constants and resources in the X11 .h files. It provides functions to create, access, and modify all the structures used by the X toolkit and Xlib. Widget class variables are also defined, and Common Lisp functions can be used as callback routines in widgets and as functions for translations.

Although it is possible to do all X and OSF/Motif-related coding in HCLX, experience during the development process showed that certain low-level X programming should be done in C++. This includes such things as initialization, color maps, and the color button.

Color Maps. The use of a graphics library like HP StarBase and the demand for high-quality shaded solid models imply the need for a private color map within the graphics windows of HP PE/SolidDesigner. When the graphics window or its top-level shell window is focused, the graphics color map is installed (copied into the display hardware) by the X window manager. On displays that support only one color map in hardware (most of the low-end and old displays), everything on the entire screen is displayed using the installed color map. When a private color map is installed, all windows using the default color map take random colors.

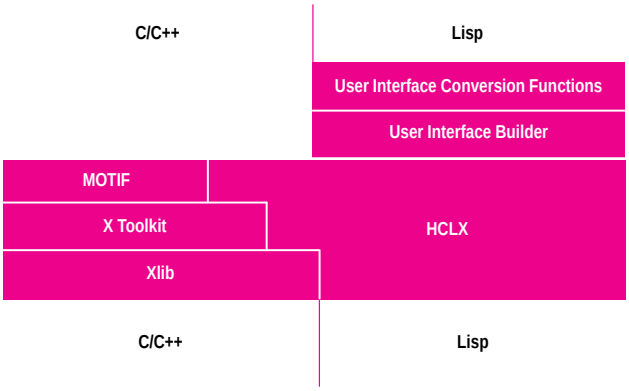


Fig. 11. Tools used to develop HP PE/SolidDesigner's user interface. HCLX is a specially created Common Lisp interface to Xlib, the X toolkit, and OSF/Motif widgets.

As soon as a window using the default color map gains the focus, the default color map is reinstalled, and the graphics windows with their private color map will have random colors. As the current color map switches back and forth from default to private, the user sees color flashing. To avoid this for the user interface of HP PE/SolidDesigner, a private color map is used for the user interface windows that has the same entries as the color map used for graphics. Along with the color map, a color converter is installed that for a given X or OSF/Motif color specification tries to find the best matching color within the color map.

Color Button. For the light settings commands, a color editor is required to give the user feedback on the colors used in the graphics windows. Therefore, a color button widget was inherited from OSF/Motif's drawn button. The color button has a small StarBase window in which colors are rendered in the same way as in the graphics windows.

User Interface Builder

HP PE/SolidDesigner's user interface builder was created using HCLX. During the prototyping phase for the user interface it became obvious that it is too expensive to train every application engineer in the basics of the X Window System and OSF/Motif. The user interface builder hides X and OSF/Motif details from the application engineer and offers facilities to create a subset of the OSF/Motif widgets.

Unique Naming. OSF/Motif widget creation procedures return a unique ID for a widget, which must be used whenever a widget is modified or referenced by some other procedure. The user interface builder changes this. Widgets are identified by unique names. These names can be specified or created automatically. The user interface builder ensures the uniqueness of the names.

Properties. For every widget only a small subset of its original resources are made available. To distinguish these resources from the full set of resources, they are called *properties*. A user interface builder property consists of a name and a corresponding value. The name is derived from the original OSF/Motif resource name by removing the prefix XmN. For example, XmNforeground becomes foreground. Some of the widget's callbacks are offered as properties. Callback properties have as a value a Lisp form, which will be evaluated when the callback is triggered. The user interface builder

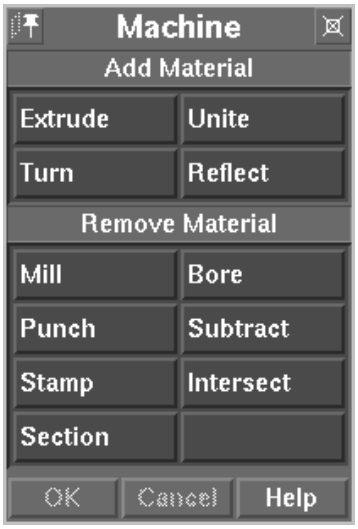


Fig. 12. Command dialog box created with a call to create-right-menu-dialog.

ensures that Lisp errors within these forms are trapped and handled gracefully. After a property has been specified for a widget, its value can be queried and the user interface builder will return the Lisp form that was used for the specification. This means that specifying red or #FF0000 as a value for the property background will result in a return of red or #FF0000 and not just a pixel value as in OSF/Motif.

User Interface Builder Action Routines. All user interface builder commands are offered as action routines. They make heavy use of the property decoders to detect input errors such as wrong property names or values. There are user interface builder commands to create widgets, modify and query widget properties, display, hide, and position widgets, and access the graphics widgets.

User Interface Convenience Functions

The user interface convenience function level is located on top of the the user interface builder level (see Fig. 11). While all the user interface builder functions are closely related to OSF/Motif, the user interface convenience functions are more abstract and not related to any window system. This level allows the programmer of a new command a fast and easy-to-use implementation of the command's user interface. The functions guarantee that the new command fits the look and feel of HP PE/SolidDesigner's user interface.

The function create-right-menu-dialog is used to create standard HP PE/SolidDesigner menus which generally appear on the right side of the user interface. The base of every right-menu dialog is a dialog shell. This allows moving and positioning these menus anywhere on the screen. A right-menu dialog can be constructed top-down with various elements. Only its width is limited to the size of two standard buttons. Fig. 12 shows a typical HP PE/SolidDesigner command dialog constructed with a call to create-right-menu-dialog.

With the function create-options-block, typical HP PE/SolidDesigner user interface objects for command options can be created. An option block can never be without a parent widget. This means that the function create-options-block doesn't

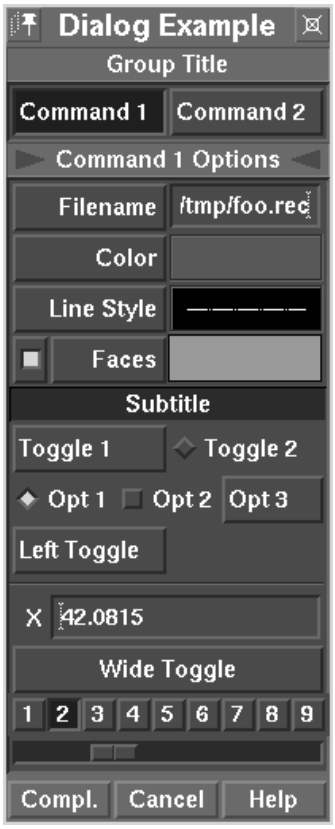


Fig. 13. Some heterogeneous option types that can be created with create-options-block.

create a dialog shell as a basis, but a form widget, which is realized in a parent widget (generally an empty form widget, also called a communication form in this article). Fig. 13 shows some of the possibilities out of which a heterogeneous option block can be constructed. Each option block has an optional title, a main part underneath the optional title, and an optional suboption form, an empty form widget below the main part as a placeholder for suboption blocks.

The function create-dialog-shell creates an empty HP PE/SolidDesigner standard dialog shell in any size. Possible elements are pin, title, close, OK, cancel, and help buttons. The empty main form can be filled with any user interface objects, which can be created using standard user interface builder calls. This function is used to create nonstandard menus such as browsers, the color editor, and so on.

Conclusion

The effort put into the development of HP PE/SolidDesigner's user interface was a good investment. The user interface is one of our key competitive differentiators. Customers like the clear structure, ease of use, and ease of learning. The Lisp-based implementation allows broad customization possibilities. The powerful concepts of HP PE/SolidDesigner's user interface and its technology provide a firm foundation for future developments.

Enhancements in Blending Algorithms

This article describes a rounding operation for a 3D CAD boundary representation (B-Rep) solid model. Complex combinations of convex and concave edges are handled predictably and reliably. At vertices the surfaces are smoothly connected by one or more surface patches. An algorithm for the creation of blending surfaces and their integration into the model is outlined. The sequence of topological modifications applied to the solid model is illustrated by examples including some special case handling.

by Stefan Freitag and Karsten Opitz

Apart from the basic Boolean operations, a modern solid modeling CAD system needs to provide easy-to-use facilities for local modifications of the primary model. One of the most important examples is the *blending* or rounding of edges, in which a sharp edge of the model is replaced by a surface that smoothly joins the two adjacent faces (see Fig. 1).

Blending surfaces serve several purposes in mechanical designs, including dissipating stress concentrations and enhancing fluid flow properties. In addition, some machining processes do not permit the manufacture of sharp edges. Smooth transitions between surfaces are also often required for aesthetic reasons. Besides functional requirements, edge blending is conceptually quite a simple operation, which makes it very popular among designers using CAD systems.

A common characteristic of almost all applications is that the smoothness of the blend is more important than its exact shape. For the user this means that it should be possible to create a blend by specifying only a few parameters. It is then the system's task to fill the remaining degrees of freedom in a meaningful manner.

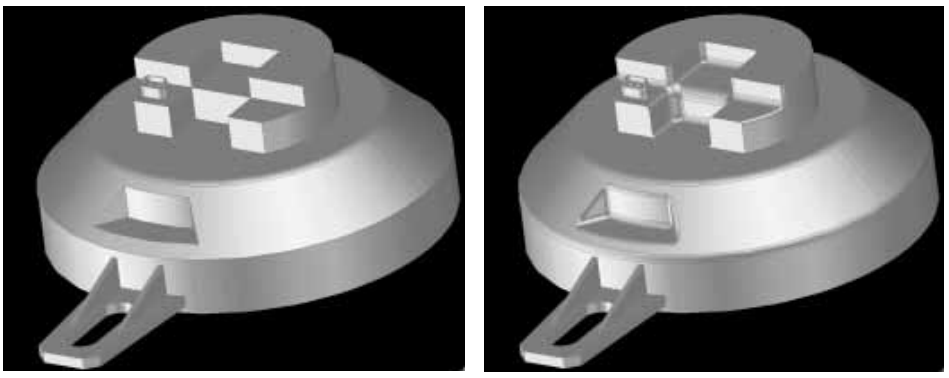
From an algorithmic point of view, blending one or more edges of a solid model simultaneously falls into two subtasks. The first is to create a surface that provides the transition between the adjacent surfaces defining the edge. Secondly, the surfaces need to be trimmed properly and integrated into the body such that a valid solid model is maintained. While the first step is a purely geometric problem, the second one involves both geometric and topological operations.

The blending module in HP PE/SolidDesigner was designed with the goal of allowing blending of a wide variety of complex edge combinations in a robust manner. This is accomplished through the use of freeform geometry as blending surfaces, along with quite involved geometric and topological considerations in several phases of the algorithm.

The lack of freeform surfaces was the primary reason for most of the restrictions concerning edge blending in HP PE/SolidDesigner's predecessor, the HP PE/ME30 3D modeling system. HP PE/ME30's kernel, the Romulus geometric modeler, does in fact provide more complex surfaces,[1] but these enhanced blends were never implemented in the product.

The current capabilities of HP PE/SolidDesigner's blending algorithm go far beyond HP PE/ME30 with respect to the topological situations that can be handled reliably. Moreover, the architecture of the algorithm allows the inclusion of future enhancements in a consistent manner.

It is the aim of this paper to illustrate the basic blending algorithm and to provide the reader with examples that demonstrate the complexity of the geometric and topological problems that must be solved to integrate one or more blend surfaces into a solid model. More information on this subject



(a) (b)

Fig. 1. (a) A solid model with sharp edges. (b) Edges rounded by blending.

can also be found in Woodwark[2] and the excellent survey of Vida.[3]

HP PE/SolidDesigner's underlying philosophy allows flexible modifications of the solid model in every stage of the modeling process. In the context of edge blending this means that it should always be possible to remove or modify an existing blend surface without regard to how it was created.

In the next section, the second section of this article, we introduce some terminology commonly used in solid modeling, in particular in the blending context. The third section describes the use model of edge blending in HP PE/SolidDesigner. An overview of the algorithm is given in the fourth section, followed by a more detailed discussion of its major steps. Finally, in the last section, we discuss some performance and stability issues.

Blending Module of the HP PE/SolidDesigner Kernel

Currently, the blending operation in the HP PE/SolidDesigner kernel implements what is commonly known as the *rolling ball blend*. This type of blend can easily be visualized as a ball moving along the edge and touching the adjacent surfaces (the *primary* surfaces) simultaneously. The touching loci are curves that define the boundaries of the blend surface. Depending on whether the radius of the ball is constant or varies while it is moving, we speak of *constant-radius* or *variable-radius* blends.

The geometry module of HP PE/SolidDesigner's kernel supports a number of different surface types (Fig. 2). These include the natural quadrics (planes, spheres, cylinders, and cones), toruses, and NURBS (nonuniform rational B-spline) freeform surfaces. All of the surface types are represented parametrically. The object-oriented design of the kernel allows the use of generic algorithms for general surfaces as well as special-case solutions for particular surface types.

Most algorithms such as surface/surface intersections or silhouette calculations behave considerably more stably and perform more efficiently when dealing with nonfreeform or *analytic* surfaces. Consequently, the blend algorithm tries to employ analytic surfaces whenever possible. This necessitates several case distinctions during the process of blend creation, which will be pointed out later.

Depending on the local geometry, that is, the convexity of the edge, blending an edge may involve adding or removing material. These operations are sometimes distinguished as *filleting* or *rounding*, respectively. In this article we will refer to both cases as blends.

If several edges to be blended meet at a common vertex, the blending surfaces should be joined in a smooth manner. We call these transitions *vertex regions* because they replace a vertex by a set of surfaces. In some special cases, a vertex region can be defined by a single analytic surface like a sphere or a torus. In general, however, they are defined by up to six tangentially connected B-spline freeform surfaces.

HP PE/SolidDesigner belongs to the class of B-Rep (boundary representation) modelers, in which the solid model is represented internally as a set of vertices, edges, and faces. In addition, the representation contains information about how these entities are related to each other—that is, the *topology* of the model. B-Rep modelers usually employ a restricted set of operations to perform topological manipulations of the model. The application of these *Euler operators* ensures the topological integrity of the model.

Integrating one or more blend faces into a solid involves quite a number of topological modifications and different Euler operators. We will not discuss the underlying concepts in detail here, but refer the reader to the standard sources.[4,5,6] For our purposes, it suffices to know that the blend algorithm employs these basic operators (for example, ADEV, ADED, KEV, KE) to create the new topological representation of the blended body.

The blend module also takes advantage of basic functionality provided by the geometry module of HP PE/SolidDesigner's kernel. Examples are closest-point calculations with respect to a curve or a surface. We call these operations *relaxing* a point on a curve or surface. This applies to curves or surfaces of any type. For instance, it is often necessary to relax an arbitrary point on the intersection curve of two surfaces. Since these operations are part of the kernel's generic functionality, we will not go into the details of their implementation.

Using the Blend Command

Like all of HP PE/SolidDesigner's commands, the user interface for the blend command is designed to be easy to use and require as little input as possible from the user. This is greatly facilitated by some general mechanisms used throughout HP PE/SolidDesigner's user interface such as the selection methods and the labeling feedback.

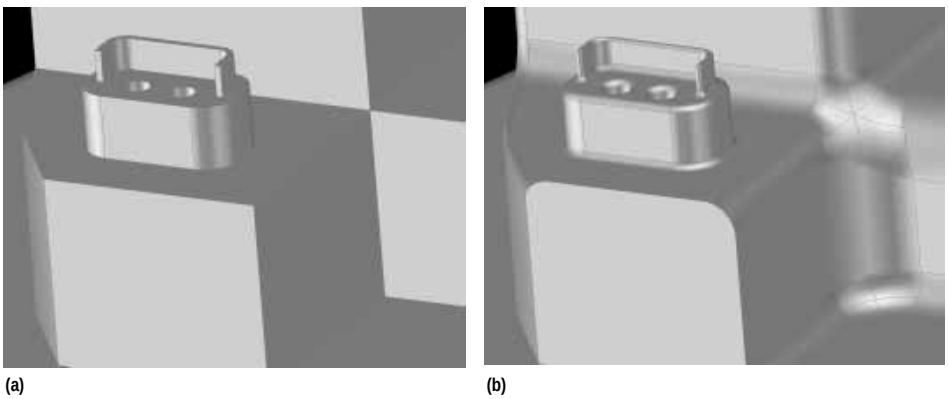


Fig. 2. Detail from Fig.1 showing different types of surfaces employed by the blending algorithm.

The blend command distinguishes two modes: the *definition mode* and the *preview mode*. In definition mode, single or multiple edges can be selected and assigned a radius (of the rolling ball). Variable-radius blends are specified by start and end radii to be assigned to the end vertices of the edge. Since the choice of the start and end vertices is arbitrary, the vertices of the currently selected edge are marked with labels. The radius of the rolling ball varies linearly between the two end vertices of the edge.

An important feature of the blend command is its ability to handle both types of blends simultaneously. This gives the user the ability to specify an arbitrary combination of constant and variable radius blends, each with possibly different radii, in a single blend session.

The blend command uses straightforward radius defaulting. For example, the constant radius of the active edge carries over to all subsequently selected edges unless the user chooses a new radius explicitly.

While processing the selected edges, the algorithm decides about the inclusion of a vertex region to provide a smooth transition between the blend surfaces. A vertex region will be created if all edges adjacent to a particular vertex are to be blended in the same session. In other words, a vertex region can easily be suppressed by blending adjacent edges one after another.

In preview mode, the blend faces are shown using a preview color. Modification of the radius or the edge information is not possible in this mode. However, upon returning to the definition mode, the user can specify further edges to be blended, modify the blend radius assigned to an edge, or remove an edge from the list.

There are two ways to terminate every command in HP PE/SolidDesigner. Canceling the blend command causes the blends to be discarded, while completing it makes the blends "real."

For convenience, the blend menu contains a small number of options:

- The part checker usually run on the blended part can be switched off to provide a faster, although possibly invalid result.
- The labels attached to edges and faces, which might be annoying if a large number of edges are selected, can be turned off.
- A *chain* option allows the user to select all edges connected tangentially to a given edge by a single pick.

Because of the complexity of the operation, blending one or multiple edges sometimes fails. While some problems are easily detected, others are caused by topological or geometrical restrictions rooted at a relatively low level. A typical example for the first kind of problem is the case where the blend radius is chosen too large. In any case, a failure is reported to the user by displaying an error message and highlighting the edge that is causing the problem.

How the Blending Algorithm Works

As noted above, the rolling ball blend provides us with a very intuitive way to define a blend surface. While moving along the edge, the ball sweeps out a certain volume. The blend surface is simply a part of the surface bounding this volume. In mathematics, surfaces that are swept out by families of moving spheres are called *canal surfaces*.[7] The cylinder and the torus are the most obvious examples.

A number of blending problems can be handled by inserting surfaces of these types. We will refer to these cases as *analytic blends*. In other than the simple cases, however, the explicit representation of a canal surface takes on quite a complicated form. Therefore, an approximation of the ideal blending surfaces by freeform blends is constructed. In particular, we use C^1-continuous B-spline surfaces.

The general algorithm is divided into a number of smaller modules. Each of these modules typically scans over all edges to be blended and performs a certain task. However, care is taken that the result is symmetric, that is, it does not depend on the order in which the edges are operated on.

The task of the first module is to filter out all cases where an analytic solution exists and flag the corresponding edges accordingly. In the second step, the touching curves of the ball with the primary surfaces are calculated. While this is straightforward for analytic blends, the boundaries of freeform blends must be computed numerically. This is accomplished by a *marching* algorithm.

Having calculated the boundaries of the blend surface, we determine their intersection points with other edges. It is often necessary to remove edges from the model to find useful intersection points. This is the first step that possibly involves topological modifications of the original body.

Other major changes to the model are done in the next two modules, which represent the blend face topologically. The first module performs the *zipping* of the original edge, that is, it replaces this edge by two new ones connected to the same end vertices. Secondly, the appropriate topology at the end vertices is inserted.

From a topological point of view, the model containing the primary blends is now complete. However, several topological entities are still without geometry. The surfaces corresponding to the blend faces, for instance, are not yet defined. These are computed in the next module based on the already available boundary data.

Furthermore, the surfaces need to be trimmed at the end vertices of the original edges. The trimming curves of the surface are, in general, computed by intersecting them with adjacent surfaces. However, it might also be necessary to intersect two adjacent blend surfaces created in the same session. The intersection curves are then "hung" under the corresponding edges.

Finally, the last major module performs the inclusion of vertex regions, both topologically and geometrically. These steps will be described in more detail later.

Analytic or Freeform Blends

It is not difficult to list all cases where a cylindrical or toroidal surface fits as a blend between the two primary surfaces. The simplest case is the one in which two intersecting planes blended by a cylindrical surface. A torus can be used when blending the edge between a conical and a planar surface as shown in Fig. 3. In a first pass over all involved edges, the algorithm tries to match one of the cases where

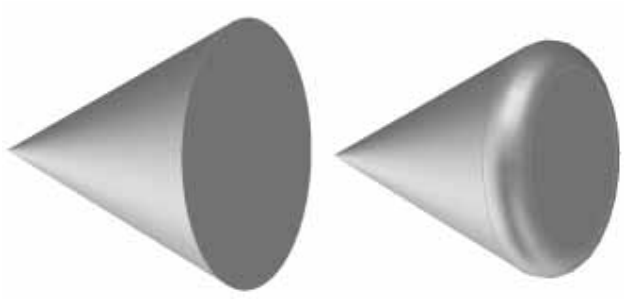


Fig. 3. A torus provides a smooth blending of the edge between a conical surface and a planar surface.

such a solution exists. The corresponding edges are then flagged as analytic.

The decision about when to employ analytic or freeform blends, however, is also dependent on other, more global factors. For example, suppose that three cylindrical blends with different radii meet at a common vertex (Fig. 4). This necessitates the inclusion of a freeform vertex region. Depending on the numerical tolerance used in the system, this might lead to very expensive B-spline surfaces in terms of data generated (the B-spline boundaries of the vertex region must lie—within some tolerance—on the adjacent cylinders). Therefore, it is often necessary to use freeform blends rather than analytic ones at a subset of the edges for the benefit of reducing overall data size. The corresponding checks are done in a second pass over the edges.

As a side-effect of switching from an analytic to a freeform blend for a particular edge, other edges adjacent to this one might be affected. This is also taken care of in the second pass.

The results of these operations are flags attached to all involved edges and their end vertices which provide information to all following modules about the types of surfaces to be used.

Blend Boundary Creation

The task of the second module is to compute the blend boundaries and tangency information along these curves. This information will be used later for the construction of the blend surfaces. The calculation of the boundaries for cylindrical and torodial blends is a straightforward exercise in analytic geometry and will not be described here. More involved and computationally more expensive is the general case, which will be the main topic of this section.

A major advantage of the rolling-ball blend is that its definition can be put into mathematical terms quite precisely. Suppose a ball with radius r moves along the edge between the primary surfaces. The curves where the ball touches the surfaces will be the boundaries of the blend surface to be inserted. The center of the ball moves along a third curve, the *spine* of the canal surface. If the radius of the ball changes while rolling, the curves touching the surfaces will define a variable-radius blend surface. In HP PE/SolidDesigner a general B-spline curve is used to define the radius function.

The spine lies entirely on a surface with constant distance r from the original surface. This is called the *offset surface*. This applies to both primary surfaces. Therefore, we can calculate the spine as the intersection curve of the two offset surfaces (Fig. 5).

Computing surface/surface intersections is a ubiquitous problem in solid modeling and many algorithms have been devised for its solution. Very popular are the marching algorithms, which trace out the intersection curve starting from a given point in its interior. In our blending algorithm, we get such a starting point by taking the midpoint of the original edge and relaxing it onto the spine. The entire curve is then computed by marching the intersection of the two surfaces in both directions. The marching stops when the curve leaves a certain 3D box provided by the calling routine. The boxes are chosen such that the resulting blend surfaces are large enough to fit into the model.

The particular strategy we employ for the marching is to reformulate the problem as one of solving a differential equation in several unknowns. The solution is then computed by a modified Euler method.

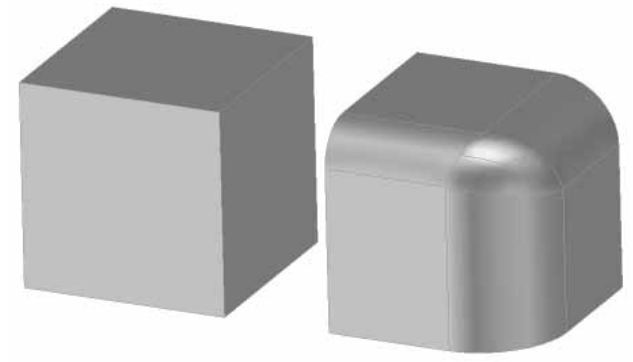


Fig. 4. Three cylindrical blends with different radii connected by a freeform vertex region.

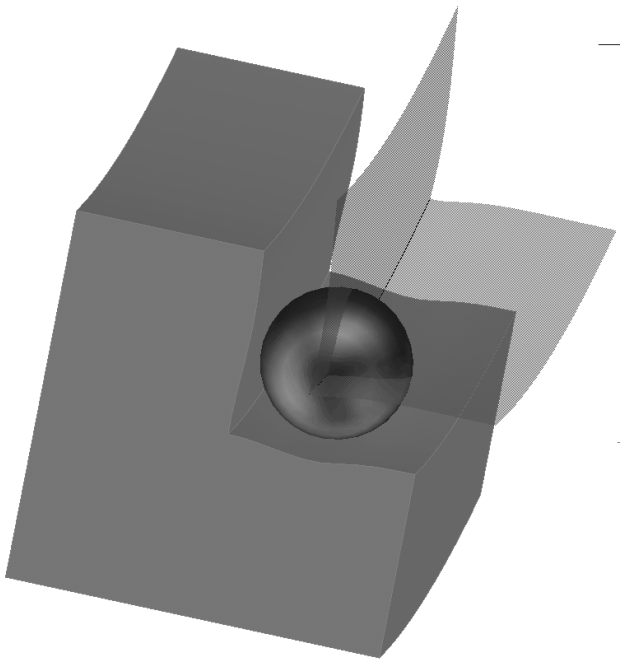


Fig. 5. The center of the rolling ball moves on the intersection curve between the two offset surfaces.

A common problem in marching algorithms is the choice of an appropriate step size. Choosing the step size too big might lead the algorithm astray. On the other hand, very small steps usually guarantee convergence of the method but might generate too much data. Therefore, we use an adaptive technique based upon the curvature of the intersection curve: a small curvature indicates that the intersection curve behaves almost like a straight line. This means that we can proceed with a large step. On the other hand, if the curve bends sharply, that is, its curvature is large, we use very small steps to capture all of its turns.

The result of these computations is a set of isolated points lying exactly on both offset surfaces and thus on the spine. Conceptually, the corresponding points on the blend boundaries can be determined by projecting these points onto the original surfaces (Fig. 6). In fact, for parametric surfaces this operation is trivial because the offset surface inherits its parameterization from the underlying surface. This means that we simply have to evaluate the primary surfaces at the parameter values of the points on the spine.

The blend boundaries are now created by constructing cubic Hermitian segments between the given points. However, we still have to check whether the entire segment lies on the surface, within a given tolerance. In cases where it doesn't, we use a fast bisection method for "pulling" the curve segment onto the surface.

While the intersection curve—and thus the blend boundaries—are traced out, we also collect tangential information along the boundaries. This information is used in the surface creation step to construct smooth transitions between the primary surfaces and the blend surface. The same bisection and representation techniques as for the boundary curves are used for these *cross-tangent curves*.

Before we conclude this section, we still have to address the question of singularities, which are critical for every marching algorithm. In our context, we have to deal with two types of singularities: those of the surfaces to be marched and those of their intersection.

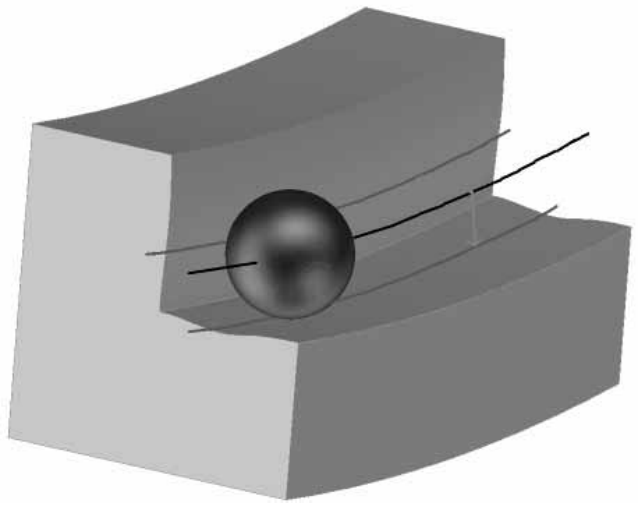


Fig. 6. The blend boundaries (red) are created by mapping the spine (black) onto the primary surfaces.

The first problem is illustrated in Fig. 7. While a small offset leads to well-behaved curves, larger distances result in offset curves with cusps or self-intersections. Analogously, we might have degenerate offsets of the primary surfaces if the distance (radius of the blend) is chosen too large. For too large a radius, a rolling ball blend is not possible. When such a situation is detected the marching stops, the entire blend algorithm stops, and the user is advised to try the operation again with a smaller radius.

The second type of singularity occurs if the primary surfaces and consequently their offsets possess a common tangent plane (Fig. 8). These *tangential intersections* typically create the biggest problems for marching algorithms. Loosely speaking, it is very difficult to find where to go at these points. However, a rolling ball blend is still well-defined. The touching curves of the ball are identical with the original edge, and the blend surface degenerates to one with zero width. HP/PE SolidDesigner's kernel enforces the rule that these extraordinary points may only occur at the endpoints of an edge. This considerably eases the task for the blending algorithm. It is quite simple to check whether the intersection curve degenerates at its endpoints. This information is provided to the routine that performs the marching. Since the algorithm starts at the midpoint of the intersection curve, the occurrence of a singular point of this type indicates that we have reached one of the endpoints of the edge.

In a final step, the segments of the boundaries and the cross-tangent curves are merged into C^1-continuous B-splines. The overall result of this module consists of four C^1-continuous curves with a common parameterization describing the boundary curves and tangency information of the blend surface.

Trimming the Blend Boundaries

After creating the blend boundaries we need to integrate the boundaries into the body. Most important, we have to find the position where the boundaries are to be trimmed. Fig. 9 shows a particularly simple example.

The six points shown in blue can be calculated by intersecting the blend boundaries with the adjacent edges at the end

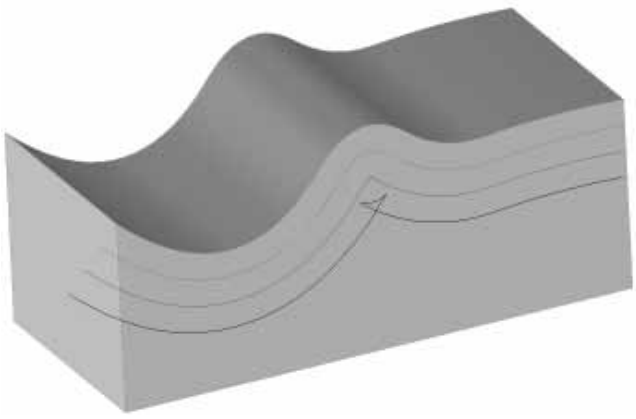


Fig. 7. When the blend radius is chosen too big, the blend boundary will have a cusp (red curve) or even be self-intersecting (black curve).

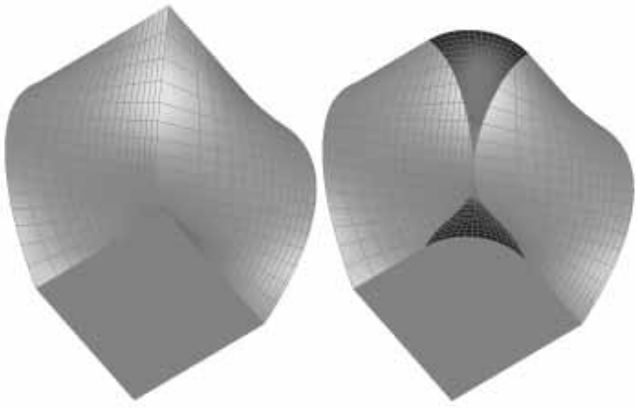


Fig. 8. If the primary surfaces have the same normal along the edge, the blend surface (blue) degenerates.

vertices. However, usually the set of edges to be blended with possibly different radii is not limited to one edge but may contain several edges or even all of them. This means that while the boundaries of a given blend face are being trimmed they must be intersected with other blend boundaries created in the same session (red points).

Intersecting a blend boundary with an existing edge of the solid model may have three results:

- One intersection point found. This is the general case.
- No intersection found. The edge is too short to be intersected by the blend boundary. In this case the edge will be removed from the model. The edge newly attached to the vertex will now be intersected by the blend boundary. Repeating this procedure guarantees the existence of at least one intersection point.
- Multiple intersection points found. Such a situation might occur, for instance, if the adjacent edge is part of a B-spline curve "wiggling" around the blend boundary. In this case, the most valuable intersection point has to be chosen. A valuable point in this context is the one that produces the most predictable and expected result.

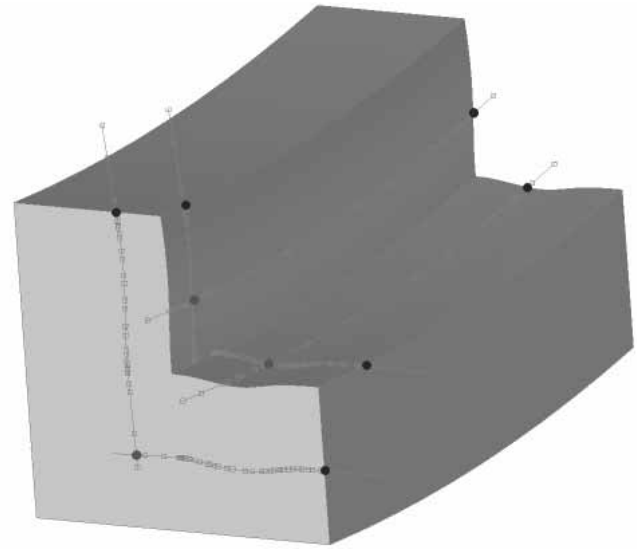


Fig. 9. The blend boundaries are trimmed at points where they intersect adjacent edges (blue) or another blend boundary (red).

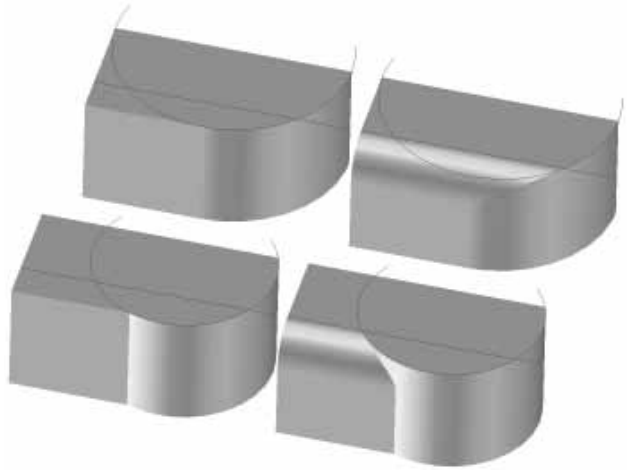


Fig. 10. Selecting the correct intersection point between a blend boundary and an adjacent edge also depends on the local surrounding geometry.

In fact, very often there are several possible solutions and all of them result in a valid solid model. Several different criteria are used to select the best intersection point. Fig. 10 shows two examples. The remaining intersection points are ignored.

Creating the Topology of the Blend Face

Having computed the trimming points of the blend boundaries, we build up the topology of the blend face. The first step is similar to opening a zipper: the original edge of the body is replaced by two new ones connected to the same vertices. The new face is then extended at its end vertices. More precisely, four new edges—two at each end—are added. In addition, the adjacent edges are split at the four trimming points (Fig. 11).

Blend Surface Creation

Now the face is ready for the inclusion of the blend surface. There are two possibilities. In the first case, analytic surfaces are inserted based on the decision made in the first module. Possible surface types are cylinders, cones, and toruses only (Fig. 12). In all other cases a freeform surface is created. We use C^1-continuous B-spline surfaces. This surface is defined by the blend boundaries created by the marching algorithm, the tangency information along these boundaries represented by cross tangent curves, and the fact that the blend surface should have circular cross sections. Using this knowledge the surface can be created very easily. The circular cross section is approximated by a single cubic B-spline segment. Although not precise, this approximation is sufficiently good for practical purposes. In fact, given the input

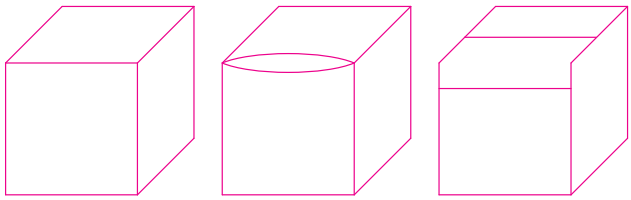


Fig. 11. Creating the topology of a blend face.

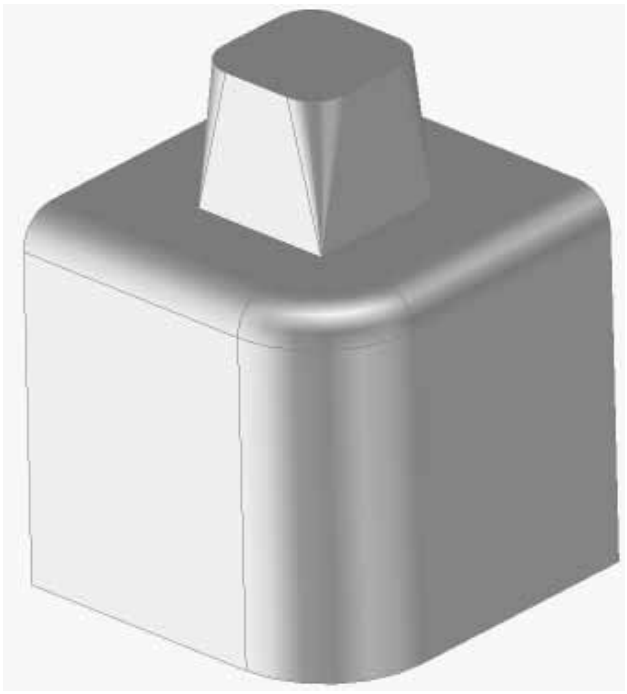


Fig. 12. A model containing only analytic blend surfaces: cylinder, cones, and toruses.

data for the cross section—boundary points and tangent directions—we use an optimal approximation based on a method described by Dokken.[8] The boundary curve information and parameterization transfer directly to the surface (Fig. 13).

Trimming the Blend Surfaces

The last step in integrating the blend surface into the solid model is to trim it at the ends. The goal is to keep the trim area as simple as possible.

Unfortunately, the authors of many edge-blend algorithms assume that they are dealing with a trimmed-face surface model and they offer no suggestion about what to do at the ends of the edge to be blended. The topological and geometrical issues are quite complex, especially when multiple edges meet at a common vertex.

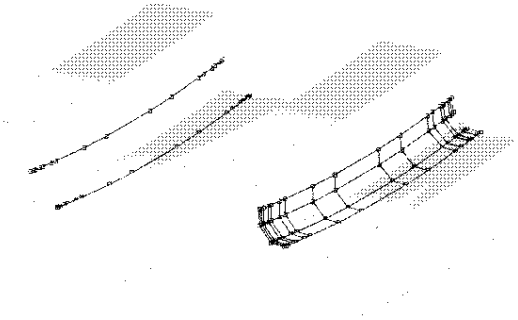


Fig. 13. Creating the geometry of a freeform blend surface: the control polygon of the blend boundaries (left) and the resulting blend surface (right).

The simplest type of termination issue arises when there is only one edge to be blended. Both boundaries must be joined at the ends of the blend face. The easiest way to do this is to intersect the blend face with all edges and faces connected to its end (Fig.14). Intersecting the blend face with these edges creates intersection points which are to be connected to form the boundary of the blend face.

The intersection points are calculated by curve/surface intersections between the blend surface and the curves of the edges at the end of the blend face. In general, a curve/surface intersection will result in multiple intersection points. In this case, the one chosen is the one closest to the vertex of the edge to be blended at this end.

If there is no intersection point of the blend face and an edge this edge is removed from the model using the Euler operator KEV. If this edge is the last one of its face, the face is removed using the Euler operator KBFV. Removing an edge means disconnecting it from its vertices and filling the gap by connecting other edges to these vertices. The newly connected edges have to be intersected with the blend face, too. However, if one of these edges is also to be blended, an intersection between its blend boundaries and the blend face is calculated. The intersection points are then connected by intersection tracks of the blend surface and the adjacent ones. In general, the result of this surface/surface intersection calculation is a set of intersection tracks. Tracks that do not contain the intersection points described above are filtered because they are not needed. The remaining tracks are sorted by the distance between two intersection points. The shortest arc is the one chosen because it minimizes the trim area at this end.

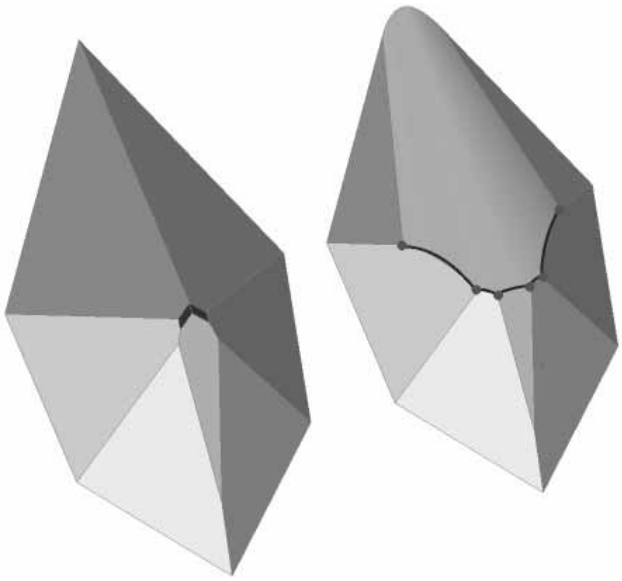


Fig. 14. Trimming a blend surface involves a number of curve/surface intersections (red points) and surface/surface intersections (blue curves). Note how the faces marked dark red are "eaten up" by the blend face.

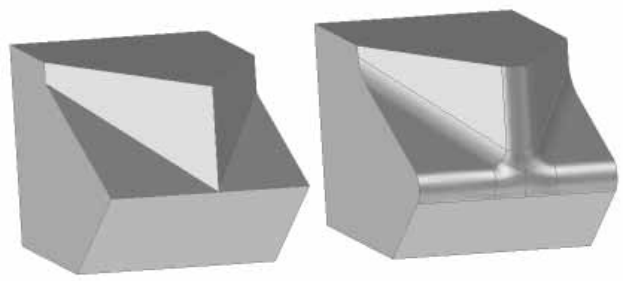


Fig. 15. When more than two edges to be blended meet at a common vertex, a vertex region is inserted to connect the blends smoothly.

Vertex Regions

A totally different situation occurs when more than two edges meet at a common vertex. In this case a set of additional faces and surfaces must be created to build a transition patch that smoothly connects all of the blend faces meeting there. This set of faces is called a vertex region (Fig. 15).

In some special cases a vertex region has only one face, which is an analytic surface (sphere or torus). In general, however, a vertex region will contain three or more faces. In HP PE/SolidDesigner the number of faces in a vertex region is currently limited to six.

Topology of Freeform Vertex Regions. At a vertex where five edges to be blended meet each other, the topology shown in Fig. 16a arises after extending the blend faces as described above. The blending algorithm transforms this topological situation by integrating five faces, each having four edges, as shown in Fig. 16b. Transforming the topology requires the use of the Euler operators KEV, ADEV, and ADED to kill an edge, add an edge, and add a whole face. Fig. 17 shows the sequence of Euler operators.

Topology of Analytic Vertex Regions. When a sphere or torus fits a vertex region the topology is changed in another way. Instead of the "star" where the blend faces meet, a single face will be created using KEV and ADED, as shown in Fig. 18. Fig. 19 illustrates the algorithm, showing the transformation step by step.

Geometry of Freeform Vertex Regions. After creating the topology of a vertex region, the corresponding geometry must be constructed and integrated. To provide a smooth transition,

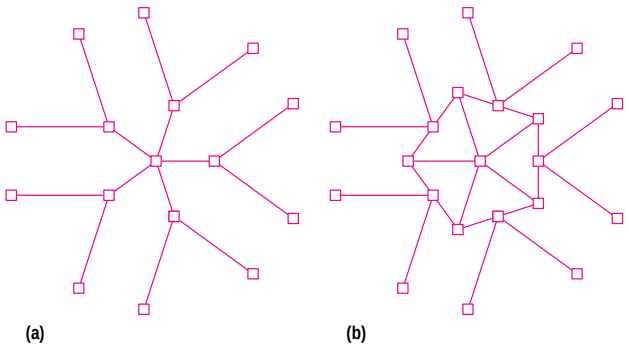


Fig. 16. (a) Topology of a vertex region where five faces meet after extending the edges. (b) Topology created for the representation of the vertex region.

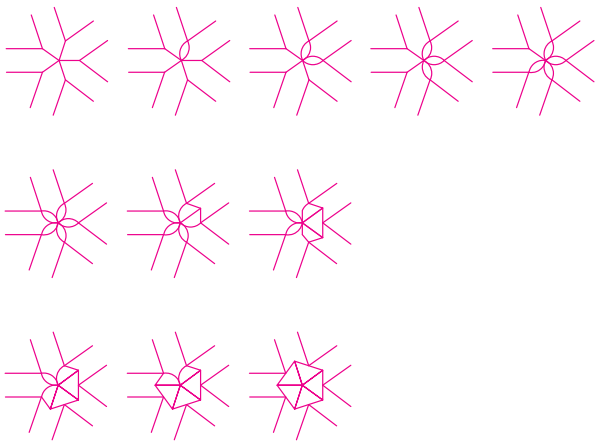


Fig. 17. Sequence of Euler operators used to transform the topology of Fig. 16a to the one of Fig. 16b .

the surfaces must satisfy two constraints. First, their boundaries must match the ones of the adjacent surfaces. Secondly, the vertex regions and the blend surfaces should possess the same tangent planes along their common boundaries. The construction of vertex regions satisfying those constraints is a classical problem in geometric modeling.[9] Among the many solutions, we mention the one proposed by Charrot and Gregory.[10] They fill a vertex region by a procedurally defined surface, that is, a surface that does not have an analytic mathematical representation but rather is defined by a method of generating it. Since the geometry kernel of HP PE/SolidDesigner does not support this type of surface, we employ an algorithm that generates a set of four-sided B-spline surfaces. More precisely, for filling an n-sided hole, we use n B-spline surfaces of polynomial degree 6 in both parameter directions.

Geometry of Analytic Vertex Regions. From the geometrical point of view analytic vertex regions are quite easy to compute because only one surface is needed and the surface type will be either a sphere or a torus.

Transition Curves. When large radii are combined with very small radii, a vertex region can look very strange, deformed, or even self-intersecting, like the left solid in Fig. 20. In such cases, instead of a three-sided vertex region, a four-sided one is used, giving a result like the right solid in Fig. 20. In general, an (n+1)-sided region is used instead of an n-sided region. This is done by introducing a transition curve between two boundaries sharing the same face. The transition curve is used whenever an intersection of two boundaries is

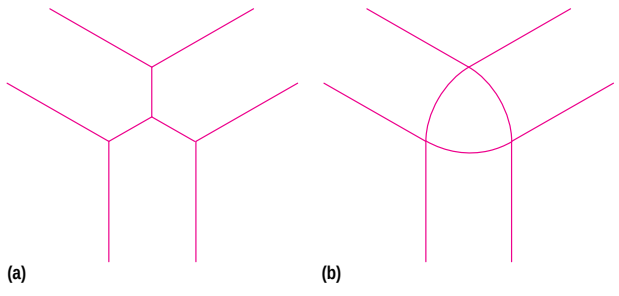


Fig. 18. When part of a sphere fits as a vertex region, a single face is created.

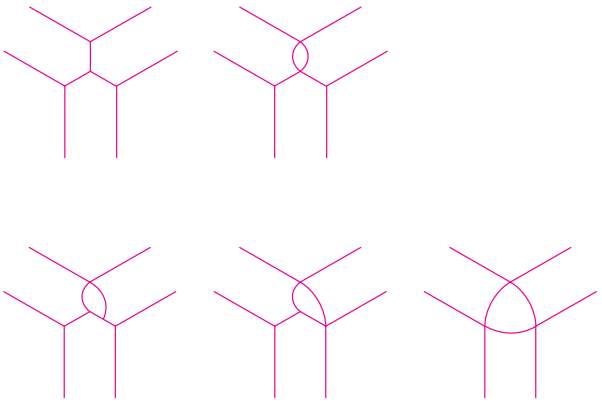


Fig. 19. Sequence of Euler operators used to create the face of Fig. 18.

"behind" the direct connection of its neighboring intersection points.

Geometrically, the transition curve is a B-spline curve defined at its endpoints by tangency conditions to both boundaries and in between by a tangency condition to the corresponding face. This curve is created using an adaptive curvature-controlled bisection algorithm similar to the one used to create the blend boundaries. The endpoints of the transition curves are constructed such that the cross section of the resulting blend surface is an isoparametric of this surface. In certain cases it is also necessary to insert a transition curve to smoothly connect two nonintersecting adjacent blend boundaries. Fig. 21 shows an example.

Special Cases

A reliable blending algorithm must be able to handle various topological and geometrical special cases predictably. Four major special cases are tangential intersections, apex creation, a singularity at the end of a blend surface, and closed curves.

Tangential Intersections. Real-life solid models often contain edges connected tangentially at a vertex to another edge. Blending these edges will result in very complex and time-consuming surface/surface intersections in the process of trimming the blend faces at the common vertex, especially when their radii differ only slightly.

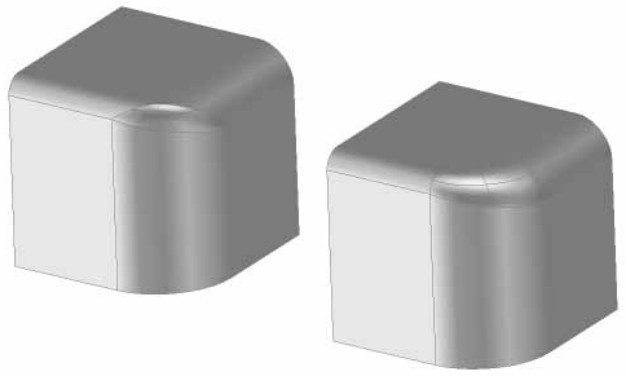


Fig. 20. When the vertex region would be too badly deformed, an additional transition curve is inserted to provide a smoother transition.

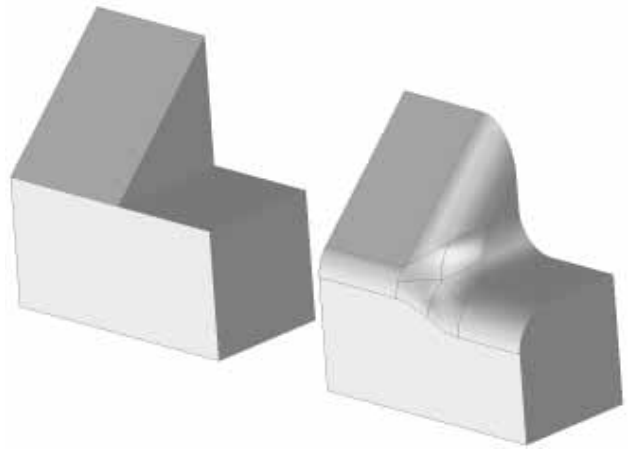


Fig. 21. A transition curve (lower edge of the vertex region) is also inserted when two adjacent blend boundaries around a vertex region don't intersect.

If two boundaries are tangential to others, the intersection point calculation is numerically very unreliable and expensive. In addition, both blend surfaces share a common region of partial coincidence, so the intersection track calculation is even more expensive than the intersection point calculation.

To avoid these problems, two edges to be blended are handled in a totally different way. No curve/surface or surface/surface intersections need to be calculated. Rather, an additional face is created that smoothly connects the two surfaces (Fig. 22).

Apex Creation. If an edge to be blended is concave, material is added to the solid model. This means that other edges become longer and faces become larger, and sometimes a singular point moves into a face. HP PE/SolidDesigner requires a topological entity, a vertex, right at the apex in this case. Therefore, after creating the blend face the required vertex is added (Fig. 23).

Singularity at the End of a Blend Surface. Sometimes at an endpoint of the edge to be blended the surface normals of the adjacent surfaces are equal—for example, two cylinders with the same radius intersected orthogonally (Fig. 24). In this case both boundaries of the blend surface meet at a common point where both surface normals are equal. There is

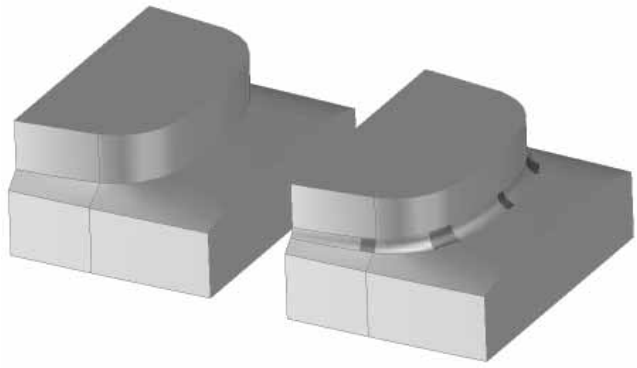


Fig. 22. Additional faces (red) are inserted where adjacent blends are tangentially connected.

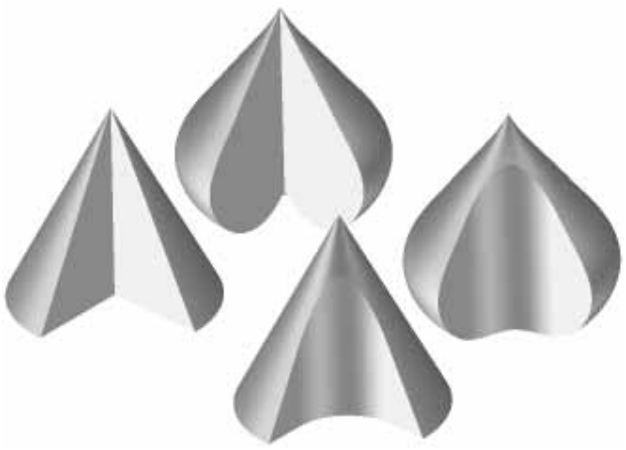


Fig. 23. When material is added by blending an edge, an apex might move from the boundary to the interior of a face.

no need to execute the trimming part of the blend algorithm because the solid is already closed at that end.

However, from the geometrical point of view, the blend surface is degenerate. One side or in this case both sides of the blend surface are degenerate isoparametric boundaries. This means that evaluating any parameter space point at this surface boundary results in the same object space point. This object space point is the position where both blend boundaries meet and the adjacent surfaces have the same surface normal.

Data Size and Performance versus Accuracy

The size of the data structures that represent freeform geometry mainly depends on the number of control points defining the curve or surface. In practice, curves can have hundreds of control points and surfaces many more. As an example, let's consider a medium-size surface with 500 control points with three coordinates each. In double-precision format, such a surface requires $3 \times 8 \times 500$ bytes or approximately 12K bytes of memory. In fact, a real-life model may contain many freeform surfaces. It is therefore quite important to reduce both the number of such surfaces and the number of control points used to represent them.

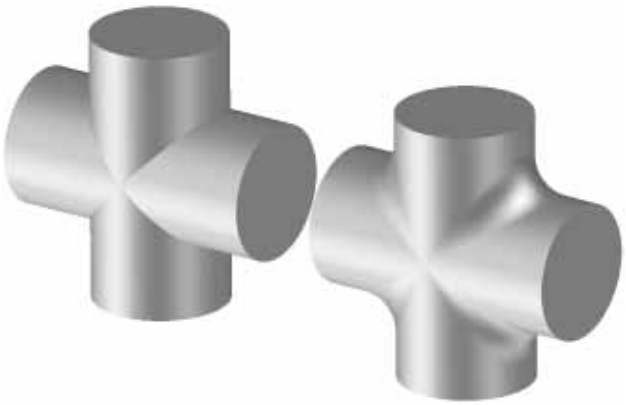


Fig. 24. Another example of two adjacent surfaces that have the same normal at an endpoint of an edge to be blended. The trimming part of the algorithm is not needed.

The size of a freeform blend surface is basically determined by the complexity of its boundaries. Boundaries with n control points lead to surfaces with 4n control points. Consequently, it is critical to generate approximations of the "true" blend boundaries with a minimal amount of data. On the other hand, the creation of the blend boundaries is one of the major factors determining the algorithm's overall performance. Finding an acceptable compromise between the conflicting requirements of speed and quality of the solution is an important design decision in the algorithm.

The same applies to the surfaces used for filling the vertex regions. The size of such a surface is quadratically dependent on the size of its boundary curves. Let's again consider an example. Assume that the boundary curves of a three-sided vertex region are general intersection curves between the primary blend surfaces and planes. It is not uncommon for approximations of those curves to contain 50 control points (HP PE/SolidDesigner works with an accuracy of up to 10^{-6}). This would lead to a vertex region of $3 \times 25 \times 25 = 1875$ control points (three surface patches of 25×25 control points each), requiring $3 \times 8 \times 1875$ bytes or approximately 44K bytes of data. Clearly, this is unacceptable for nontrivial models.

There are several possibilities for reducing the amount of data. The most critical factor is the approximation tolerance used in the system. For example, reducing the accuracy from 10^{-6} to 10^{-3} typically reduces the size of freeform data structures by a factor of ten. Not only are the geometric calculations speeded up considerably when using a lower accuracy but also the overall performance of the system is improved because of the reduced demand for memory management. HP PE/SolidDesigner offers the user the ability to select the accuracy in a range of 10^{-2} to 10^{-6}. This allows the user to choose between high-precision modeling and a faster but less precise approach.

Secondly, the handling of special cases can reduce the amount of data significantly. Let's again take a look at freeform vertex regions. If the primary blend surfaces are created such that the boundaries of the vertex regions are isoparametric curves of the primary blends (the procedure for doing this is beyond the scope of this article), the 50 control points can be reduced to 4. The vertex region will then contain $3 \times 7 \times 7 = 147$ control points (the additional control points along the boundaries—seven rather than four—are the result of the mathematical construction), for a total of approximately 3.5K bytes.

Another example is the trimming of a blend face. In this step a number of surface/surface intersections must be calculated. In general, an intersection of two surfaces will result in not only one curve, but several intersection points, curves, or even surfaces. However, in the blending context there is important knowledge about the blend surface and the face it intersects. At least one and in some cases two points on the intersection track are known from the preceding curve/surface intersections. Providing these points as "seeds" to the intersection routines increases both the speed and the reliability significantly. In addition, boxes in the parameter space of the surface are used to limit the calculation of intersection information to regions that are of interest.

From these examples we see that the good overall performance of the algorithm is mainly guaranteed by appropriate special case handling at critical points. In fact, a large portion of the code in the blending module was developed to deal with these situations.

Acknowledgments

The development and implementation of the blending algorithm in its early versions was largely conducted by our former colleagues Hermann Kellermann and Steve Hull. A part of the code was developed at SI/Sintef in Oslo, Norway.

References

1. A. Rockwood and J. Owen, "Blending Surfaces in Solid Modeling," in *Geometric Modeling: Algorithms and New Trends*, G. Farin, ed., SIAM, 1987, pp. 367-384.

2. J.R. Woodwark, "Blends in Geometric Modeling," in *The Mathematics of Surfaces II*, R.R. Martin, ed., Oxford University Press, 1987, pp. 255-297.

3. J. Vida, R.R. Martin, and T. Varady, "A survey of blending methods that use parametric surfaces," *Computer-Aided Design*, Vol. 5, no. 5, 1994, pp. 341-365.

4. B. Baumgart, *Geometric Modeling for Computer Vision*, PhD Thesis, Stanford University, 1974.

5. I. Braid, R.C. Hillyard, and I.A. Stroud, "Stepwise Construction of Polyhedra in Geometric Modeling," in *Mathematical Methods in Computer Graphics and Design*, K.W. Brodlie, ed., Academic Press, London, 1980, pp. 123-141.

6. M. Mantyla, *An Introduction to Solid Modeling*, Computer Science Press, Rockville, 1988.

7. W. Boehm and H. Prautzsch, *Geometric Concepts for Geometric Design*, AK Peters, Willesley, 1994.

8. T. Dokken, M. Daehlen, T. Lyche, and M. Morken, "Good approximation of circles by curvature continuous Bezier curves," *Computer-Aided Geometric Design*, Vol. 7, 1990, pp. 30-41.

9. J. Hoschek and D. Lasser, *Fundamentals of Computer-Aided Geometric Design*, AK Peters, Willesley, 1993.

10. J.A. Gregory, "N-sided surface patches," in *The Mathematics of Surfaces*, J.A. Gregory, ed., Clarendon Press, 1986, pp. 217-232.

Open Data Exchange with HP PE/SolidDesigner

Surface and solid data can be imported from HP PE/ME30 and exchanged with systems supporting the IGES, STEP, and ACIS formats. Imported data coexists with and can be manipulated like native data.

by Peter J. Schild, Wolfgang Klemm, Gerhard J. Walz, and Hermann J. Ruess

HP PE/SolidDesigner supports the coexistence of surface data with solid data and provides the ability to import and modify surface and solid design data from a variety of CAD systems. Backward compatibility with HP PE/ME30 preserves the investment of existing HP customers. Using improved IGES (Initial Graphics Exchange Standard) import capability, both surface and wireframe data can be imported. Surface data and solid data can also be imported and exported using the STEP (Standard for the Exchange of Product Model Data) format. Once imported, this data can coexist with HP PE/SolidDesigner solid data. It can be loaded, saved, positioned, attached to, managed as part and assembly structures, deleted, and used to create solids. Attributes such as color can be modified. If the set of surfaces is closed, HP PE/SolidDesigner will create a solid from those surfaces automatically.

HP PE/SolidDesigner 3.0 also allows solid parts and assemblies to be exported to ACIS-based systems using Version 1.5 of the ACIS® SAT file format. This feature provides a direct link to other ACIS-based applications.

From PE/ME30 to PE/SolidDesigner

HP PE/ME30 is a 3D computer-aided design (CAD) system based on the Romulus kernel. To preserve the investment of existing customers it was required that the transition from HP PE/ME30 to HP PE/SolidDesigner be as smooth as possible. Therefore, an HP PE/ME30 file import processor is a integral component of HP PE/SolidDesiger.

In HP PE/ME30, 3D objects are built from analytic surfaces like cylinders, cones, spheres, planes, and toruses. The intersections of these surfaces can be represented as explicit analytic curves such as straight lines, circles, and ellipses, or implicitly by describing the surfaces involved and providing an approximation of the intersecting arc. Parabolic and hyperbolic intersections are represented implicitly.

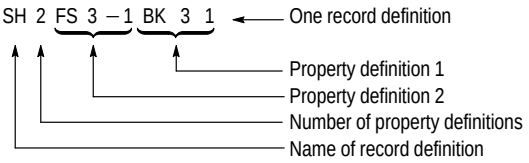
HP PE/ME30 Native File Organization

HP PE/ME30 supports the Romulus textual transmit format. The transmit file is not intended to be read by humans but the general structure can be examined. The file contains only printable characters, and real values are represented as text strings. The full format of a transmit file consists of six different sections. These will be described using the example of a single cylinder positioned at the origin of HP PE/ME30's coordinate system with base circle radius 10 and height 20.

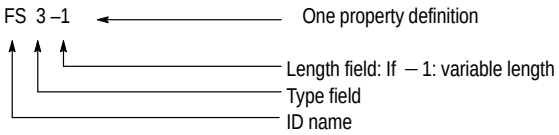
The first section, the header section, describes the environment, the machine type, the user login of the file creator, and the time and date when the model was created.

```
@* AOS
@* Machine type HP-UX
@* Transmitted by user_xyz on 27-May-94 at 13-06
```

The second section contains index and counting information related to the schema described in the third section. The schema defines the data structures used to represent the objects. It consists of a collection of record definitions. The following is an example of a record definition:



The following is an example of a property definition:



In the second section of the transmit file the number of record types, the numbers of record instances and property instances, the name of the schema, and its version and update number are supplied. The record instances and property instances contain the concrete data describing the model. The semantics and the sequence of data entities have to conform to the format specified by the corresponding record definition and property definition entities.

The information in the cylinder example file says that 11 instances of record definitions are supplied to describe the schema for the instance of the cylinder. For the actual object, 23 record instances built out of 115 property instances are used.

* A kernel is the heart of a modeling system. Currently, three kernels are used in various CAD systems. These are Romulus from Shape Data, Parasolid, an extension of Romulus, and the ACIS Kernel from Spatial Technology.

```
1
11
23 115
ROMDSCHMA 7 4
16
```

The third section, the schema section, contains the definition of the data structures used to represent the model. This section consists of the subset of record definitions from the HP PE/ME30 internal data structure schema that are needed to represent the model. The schema sections of files representing different models will be different. The schema section for the example cylinder is:

```
BY  19 UP 4 −1 SE 3 −1 TX 5 –36 FI 4 −6 CI 4 −12 PI 4 −8
GI 4 −1 SI 4 −3 TI 4 −3
RA  2 1 RN 2 1 ZI 4 −2 FN 1 1 CN 1 1 PN 1 1 TN 1 1
SN 1 1 ZN 1 1 NM 1 1
SH  2 FS 3 –1 BK 3 1
FA  8 UP 4 −1 AK 3 −1 RV 1 1 SF 3 1 SX 2 1 VR 3 −1
HA 2 −3 SL 3 1
VR 4 PT 3 1 BE 3 1 BV 3 1 FE 3 1
ED 2 CU 3 1 RV 1 1
CU 3 UP 4 −1 AK 3 –1 TR 3 1
TR 6 UP 4 −1 AK 3 −1 BK 3 1 EQ 2 −7 TS 3 2 TY 1 1
PT 4 UP 4 –1 AK 3 −1 CO 2 −3 GP 3 1
GP 5 UP 4 −1 AK 3 −1 BK 3 1 CO 2 −3 PX 3 1
SF 7 UP –1 SD 3 −3 AK 3 −1 BK 3 1 EQ 2 −7 SU 3 −5 TY 1 1
UA 3 OW 3 1 CL 1 1 II 1 *1
```

The fourth section contains, for each record type defined in the schema section, the number of data objects used for the transmission of the model. The sequence of numbers is identical to the sequence of record definitions used in the schema section. In the cylinder example, the object consists of one body built of one shell built of three faces. Four vertices, four coedges, two curve geometries, two edges, two points with two geometric point definitions, three surfaces, and one attribute are needed to represent the cylinder object. The file contents are:

```
1 1 3 4 4 2 2 2 2 3 1
```

The fifth section, the data section, contains the data structure instances. The contents of all records needed to represent the object are found in this section. To every record an integer record label is assigned. This number will be used in other record instances to point to the instance. In general the instances in the file appear in the order in which they are referenced by other entities. The data of an entity instance is not split. If forward references are contained in the instance definition the next instances can be found in exactly the same sequence as referenced. Because this rule applies recursively, newly referenced entities can be found first in the physical file sequence. If all references of an entity are resolved completely the next reference of the next higher level will be resolved. For the cylinder, the data section is:

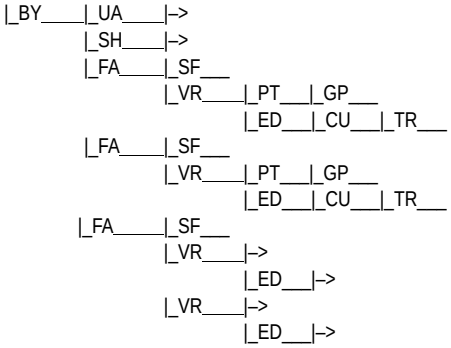
```
1
1 1 25 Color 1 2 0 3 3 F0 4 F1 5 F2 2 14 E0 15 E1 2 18 P0 19 P1 0 0 0 0.000001
0.00000000001 0 3 3 2 0 0 0 0 25 1 1 1 16777215 2 0 1 3 0 0 0 22 0 1 6 0 2 22
0 0 0 0 6 0 0 0 0 0 −1 0 1 6 18 10 6 10 18 0 0 0 20 20 0 0 0 3 0 10 0 18 10 14
0 14 0 0 16 16 0 0 0 7 0 0 0 0 0 −1 10 0 0 2 4 0 0 1 23 0 1 7 0 2 23 0 0 0 0 6
0 0 20 0 0 −1 0 1 7 19 11 7 11 19 0 0 0 21 21 0 0 0 3 0 10 20 19 11 15 1 15 0
0 17 17 0 0 0 7 0 0 20 0 0 −1 10 0 0 2 5 0 0 0 24 0 2 8 9 0 2 24 0 0 0 0 7 0 0
0 0 0 1 10 0 2 8 18 12 8 12 12 14 1 9 19 13 9 13 13 15 0
```

The sixth and last section contains, for each top-level object transmitted in the file, the corresponding root entity and its name. In the cylinder example only one object is transmitted. HP PE/ME30 supports user-named objects, but in this example an HP PE/ME30 default name, B0, has been used for the cylinder.

```
1
B0
```

Analyzing the Transmit File

Because the information content of an HP PE/ME30 file cannot be understood by simply looking at the file, several internal analysis tools are used to extract the information. Statistics showing the number of different curve and surface types give a first hint of the complexity of the file. A graphical presentation of the data instances of a file can be generated.



This reference structure can be read easily. The (cylinder) object in the file is a body (BY) which consists of a shell (SH) and three faces (FA). Shell and faces share the same hierarchy level. Each face consists of a reference to a surface (SF) and a start vertex (VR). Each start vertex is based on a geometric definition of a point (PT) and serves as the anchor vertex of an edge loop. A loop is not represented explicitly in the HP PE/ME30 exchange file. The implicit connection is done by a reference from a start vertex to the next and previous vertices in the loop. The edge (ED) entity represents the topological direction of the edge with respect to the loop. The curve (CU) entity is an intermediate instance on the way to the curve's geometry (TR).

If complete information from the data section is needed a translation tool is available that maps the data section to a format much more useful for human readers. The following extract describes how one of the faces and the corresponding surface component of the cylinder example are represented. The mapping from the data section to the readable format is also supplied.

For this component from the data section:

```
.....................................................
  ..........5 0 0 0 24 0 2 8 9 0 2 24 0 0 0 0 7 0 0
0 0 0 1 10 0 2 .....................................
```

the corresponding translated part is:

```
5 = FA (Face owning (anchor) vertex), the properties are :
        UP is EMPTY                 ... List of permanent universal attributes
        AK is EMPTY                 ... Backpointer from element of feature
        RV :  INTEGER = 0       . .. Sense of face, edge geometry
        SF :  POINTER = 24      . .. Surface of face
        SX :   REAL   = 0       . .. Hatching pitch
    1   VR :  POINTER = 8           .. Anchor of face
    2   VR :  POINTER = 9           ... Anchor of face
        HA is EMPTY                 ... Hatch direction
        SL :  POINTER = 2           ... Shell of face

24 = SF (Surface of face), the properties are :
        UP is  EMPTY                .. List of permanent universal attributes
        SD is  EMPTY                ... Surface supporting this surface
                                        definition
        AK is  EMPTY                ... Backpointer from element of feature
        BK :   POINTER = 0          ... Backpointer from assembly or body to
                                        token
        1   EQ :   REAL   = 0       ... Geometry definition
        2   EQ :   REAL   = 0       ... Geometry definition
        3   EQ :   REAL   = 0       ... Geometry definition
        4   EQ :   REAL   = 0       ... Geometry definition
        5   EQ :   REAL   = 0       ... Geometry definition
        6   EQ :   REAL   = 1       ... Geometry definition
        7   EQ :   REAL   = 10      ... Geometry definition
            SU is   EMPTY           ... Surface supported by this suface
            TY :    INTEGER = 2 (CYLINDER)   ... Geometry type
```

Import Module

The HP PE/ME30 to HP PE/SolidDesigner import interface is linked directly to the HP PE/SolidDesigner code. In HP PE/SolidDesigner's user interface it simply adds a button to the external filing menu. If a file name is specified, the processor is activated. Internally, several C++ classes are added to HP PE/SolidDesigner to represent the schema and instance entities of the HP PE/ME30 file. For every supported HP PE/ME30 record definition entity a class derived from a generic record instance object is defined. The most important member function of each of these classes is the convert function. This function performs the mapping of the HP PE/ME30 file object to the corresponding HP PE/SolidDesigner entity.

The three main components of the HP PE/ME30 to HP PE/SolidDesigner processor are a lookup table, a schema manager, and a set of classes to represent the supported HP PE/ME30 file entities.

The lookup table is part of the interface to an HP PE/ME30 file. The main task of this table is to manage the mapping of HP PE/ME30 file entities to already created corresponding HP PE/SolidDesigner entities. A lookup table is generated for every open HP PE/ME30 file.

A schema manager is initialized if a new HP PE/ME30 file is opened. It contains the schema section information found in the newly opened file. For every open file a corresponding schema manager is available to control the interpretation of the entities of the file.

The record instance class builds the third basic data structure of the processor. Record instances are generic containers to store all of the data objects that can be expressed by valid record definitions. The constructor of the record instance class calculates the entity type from the reference number and then allocates memory and reads in the properties from the file corresponding to the property definitions of the schema. For every supported HP PE/ME30 entity a separate C++ class is derived from the record instance class, but the generic constructor is used for all subtypes. The main differentiator between the classes is the convert function.

Conversion Process

The convert function of the record instance class itself is not called by the conversion process. Rather, every derived class implements its specific conversion function (in this sense the convert function is purely virtual in C++). The individual conversion function converts itself to an HP PE/SolidDesigner entity.

Conversion and the creation of new derived instances of the record definition class constitute a recursive process. If during an active conversion an unresolved (not already converted) reference is found the corresponding HP PE/ME30 file entities can be found as the next entities in the physical file (see the description of the data section). The conversion module then creates a new derived instance of the record instance class and forces the translation of this entity to a HP PE/SolidDesigner entity that can be used to complete the conversion of the current entity. The algorithm is as follows:

```
A reference to an HP PE/ME30 file entity is found:

Already "converted"? (lookup table search)
    YES:  Use the available conversion result
    NO:   Create the new derived class of record instance
          Call the convert function
          Attach the conversion result to the lookup table
          Delete the instance to free the memory used
          Use the newly generated conversion result to continue the
            conversion.
```

Nonanalytic Intersection Curves

The conversion for intersection curves is not done on the fly, but by a postprocessor after the rest of a body is converted completely. The convert routine for an intersection track simply collects the two intersecting surfaces and all available additional information found in the file to represent the intersection. The completion of the intersection curves is done by the convert function for HP PE/ME30 bodies. After a first intermediate topology of the new HP PE/-SolidDesigner body is calculated and all analytic surfaces and analytic curves are attached to the created body, the calculation of the intersections begins.

The topology of the intersection between two surfaces in HP PE/SolidDesigner is not always the same as in HP PE/ME30 because different constraints on topology and geometry exist in the two modelers. For instance, it may be necessary to represent the single segment found in HP PE/ME30 as a sequence of different curves. In such cases the original topology has to be modified and some edges may be split. To find the appropriate intersection in HP

PE/SolidDesigner is mainly a selection process. In many cases two surfaces intersect at not only one but several distinct sections.

Consider the intersection of a cylinder with a torus in the case of perpendicular axes. Four possible intersection curves may be part of the model (see Fig. 1). In the HP PE/ME30 file additional help points are supplied to allow the correct selection. The direction of the intersection curve (the tangent to the curve) is not guaranteed to be the same in HP PE/SolidDesigner as in HP PE/ME30. Therefore the correct fit to the model is calculated and the resulting direction is reflected in the topology of the imported model.

Quality and Performance

To test the quality of the HP PE/ME30 import processor a large HP PE/ME30 test library has been compiled. It now contains more than 2300 examples of parts and assemblies. All of the test cases used during HP PE/ME30 development and support are included along with new user models consisting of recently acquired data from internal and external HP PE/ME30 users. An additional test matrix subtree was developed by creating base parts with critical features. In particular, all possible surface-to-surface intersections and various special cases have been generated.

The regression test procedure is to import HP PE/ME30 models from the test library part by part and perform the HP PE/SolidDesigner body checker operation on each. The loading time and the body checker result are collected in a reports file. A reports file can be analyzed by a shell script to supply a statistical summary of the current quality of the HP PE/ME30 interface. Because of the large amount of test data a complete test takes a long time. Therefore, an intermediate test is available. The complete test performs the basic load and check test on all currently available test models of the library directory. The intermediate test examines the reports file of the latest complete test and repeats all reported problems. It also repeats a random selection of the successful tests. At this time over 99% of the complete test conversions are classified as successful.

The performance of the import process for HP PE/ME30 files is mainly dependent on three variables: the size of the schema, the number of entities, and the number of intersections that have to be calculated:

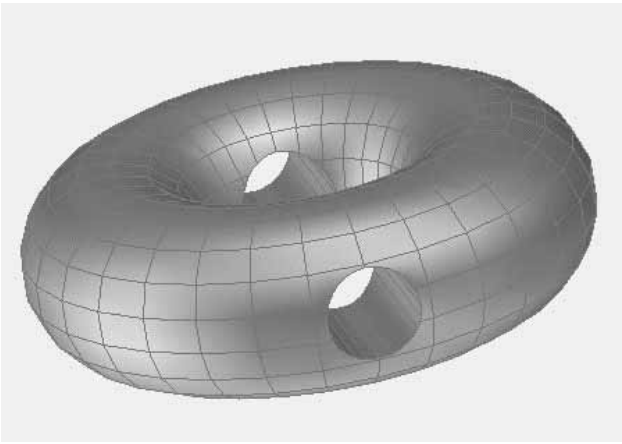


Fig. 1. Intersection of a torus and a cylinder.

$$\text{Load Time} \approx \text{Size} \times k1 + \text{Entities} \times k2 + \text{Intersections} \times k3,$$

where $k1 < k2 << k3$. The size of the schema section does not vary very much between different files and is normally relatively small compared to the size of the data section. The number of entities and the file size are strongly related. The calculation and selection of the nonanalytic intersection curves fitting the model is a relatively expensive component of the processor because a completely new representation of the data structure has to be generated.

Data Exchange Using IGES

An important task in computer-aided design is the transfer of the completed model to downstream applications and other CAD applications. These applications vary from finite element analysis and numerically controlled (NC) manufacturing to visualization and simulation. HP PE/SolidDesigner currently uses IGES 5.1 (Initial Graphics Exchange Specification) for file-based data exchange.

Because of the broad variety of receiving systems an IGES interface must be flexible so that the contents of the output file match the capabilities of the receiving system. It must be possible to transfer whole assemblies keeping the information on the parts tree, or only specific parts of a model, or even single curves or surfaces. This is achieved by a mixture of configuration and selection mechanisms.

An analysis of the IGES translators of many different systems showed that it is possible to classify them in four main categories:

- Wireframe Systems. These systems are only capable of importing curve geometry. This is typical for older CAD systems or 2D systems with limited 3D capabilities.
- Surface Systems Using Untrimmed Surfaces. These systems are capable of importing untrimmed surfaces and independent curve geometry. This is typical for low-end NC systems that need a lot of interaction to create tool paths and define areas.
- Surface Systems Using Parametrically Trimmed Surfaces. These systems are able to handle trimmed surfaces. Trimming is performed in the parametric domain of the surfaces. Periodic surfaces are often not handled or are incorrectly handled. Each surface is handled independently. This is typical for surface modelers and sophisticated NC systems.
- Topological Surface Systems and Solid Modelers. These systems are able to handle trimmed surfaces using 3D curves as trimming curves. They are able to handle periodic surfaces, nonplanar topology, and surface singularities. Connection between adjacent trimmed surfaces is maintained and the normal to the trimmed surface is important for inside/outside decisions. This is typical for advanced surface and solid modelers.

HP PE/SolidDesigner's IGES interface is designed to work in four output modes: wireframe, untrimmed, trimmed parametric, and trimmed. Each output mode represents one of the categories of receiving IGES translators. This has the advantage of giving as much information about the solid model as possible to high-end systems (trimmed, trimmed parametric), without burdening low-end interfaces with too much information. For some modes (trimmed parametric) more configuration parameters allow fine tuning to specific systems to

maximize the transfer rate. Each mode has a specific entity mapping that describes which IGES entities are used to describe the model (see Tables I, II, and III). Users can specify additional product related data and arbitrary comments for the start and global sections of the IGES file directly via the IGES output dialog box. Specific configurations can be saved and loaded so that the configuration has to be determined only once for each receiving system. Fig. 2 shows the IGES dialog menu.

To allow maximal flexibility in what is translated, the user is allowed to select assemblies, parts, faces, and edges and arbitrary combinations. All selected items are highlighted and the user can use dynamic viewing during the selection process. If the user selects assemblies, the part tree is represented with IGES entities 308 and 408 (subfigure definition and instance). Shared parts are represented by shared geometry in the IGES file.

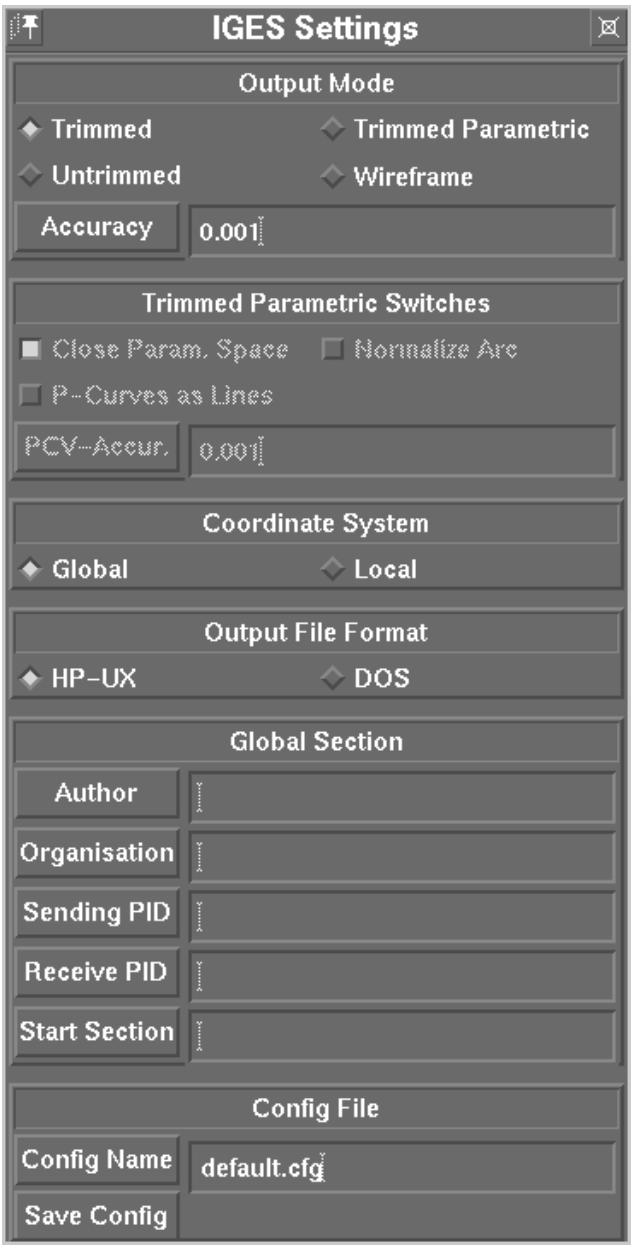


Fig. 2. HP PE/SolidDesigner IGES output dialog menu.

Table I
Curve Mapping

HP PE/SolidDesigner	IGES 3D Entity
Straight	Line (110)
Circle	Circular arc (100) with transformation
B-spline	Rational B-spline curve (126)
Intersection curve	Rational B-spline (126)
Parameter curve	Rational B-spline (126) or line (110)

Trimmed Mode

The trimmed mode is the closest description of the internal B-Rep (boundary representation) data structure of HP PE/SolidDesigner. It uses the IGES bounded surface entities 143 and 141 as the top element of the model description. Each selected face of the part maps to one bounded surface (entity 143) containing several boundaries (entity 141). Trimming of the surfaces is performed by 3D model space curves. To fulfill the requirements of the IGES specification of entities 141 and 143 some minor topological and geometrical changes of the HP PE/SolidDesigner internal model have to be made. Vertex loops are removed, propedges on toruses are removed, and intersection curves are replaced by B-spline approximations.

Because the IGES bounded surface entity 143 does not have any information about topological face normals, the surfaces are oriented so that all geometrical normals point to the outside of the part (Fig. 3). Thus, enough information is put into the IGES file that a receiving system can rebuild a solid model from a complete surface model.

Untrimmed Mode

The untrimmed mode contains basically the same information as the trimmed mode. For each face the untrimmed surface plus all trimming curves are translated. But instead of explicitly trimmming the surfaces with the appropriate entities, surface and trimming curves are only logically grouped together. This usually requires manual trimming in the receiving system, and is only suited for some special applications.

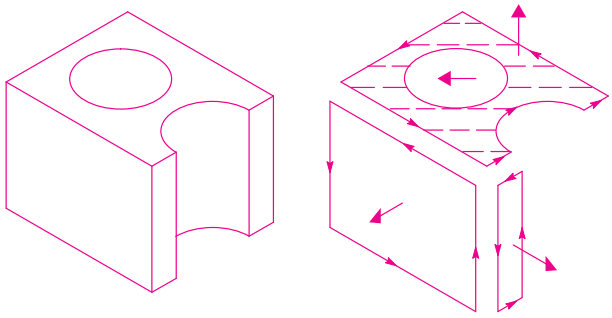


Fig. 3. (left) Solid model. (right) Surface model with normals.

Table II
Surface Mapping

HP PE/SolidDesigner	IGES 3D Entity (trimmed and untrimmed)	IGES 3D Entity (trimmed parametric)
Plane	Plane (108)	Ruled surface (118)
Cylinder	Surface of revolution (120)	Ruled surface (118)
Sphere, Torus, Cone, Spun B-spline	Surface of revolution (120)	Surface of revolution
B-spline surface	B-spline surface (128)	B-spline surface (128)
Parallel swept B-spline	Ruled surface (118)	Ruled surface (118)

Trimmed Parametric Mode

The trimmed parametric mode uses the IGES trimmed parametric surface entity (144) and the curve on parametric surface entity (142) as representations of a trimmed surface. These entities have been established in the IGES standard for a longer time than entities 143 and 141 or the trimmed mode. For this reason they are more commonly used. The main difference from the trimmed mode is that the trimming is performed in the parametric domain of the surfaces. Each surface must have a parametric description that maps a point from the parameter domain D (a rectangular portion of 2D space) to 3D model space:

$$S(u,v) = (X(u,v), Y(u,v), Z(u,v)) \quad \text{for each } (u,v) \text{ in D.}$$

$$D = \{\text{all } (u,v) \text{ with } u_{min} \leq u \leq u_{max}, v_{min} \leq v \leq v_{max}\}.$$

The following conditions apply to D:

- There is a continuous normal vector in D.
- There is a one-to-one mapping from D to 3D space.
- There are no singular points in D.

Furthermore, trimming curves in 2D space must form closed loops, and there must be exactly one outer boundary loop and optionally several inner boundary loops (holes). Fig. 4 illustrates parameter space trimming.

These restrictions make it clear that there will be two problem areas when converting HP PE/SolidDesigner parts to a parametric trimmed surface model: periodic surfaces and surface singularities.

On full periodic surfaces like cylinders, HP PE/SolidDesigner usually creates cylindrical topology. There will not necessarily be exactly one outer loop. Furthermore, 3D edges can run over the surface seam (the start of the period) without restriction. This leads to the situation that one edge may have more than one parametric curve (p-curve) associated with it. Also the p-curve loops may not be closed even if the respective 3D loop is closed. Fig. 5 illustrates this situation.

HP PE/SolidDesigner avoids this problem by splitting periodic surfaces along the seam and its antiseam. The seam and antiseam are the isoparametric curves along the parameters u_{min} and $u_{min}+u_{period}/2$. Thus, one face may result in

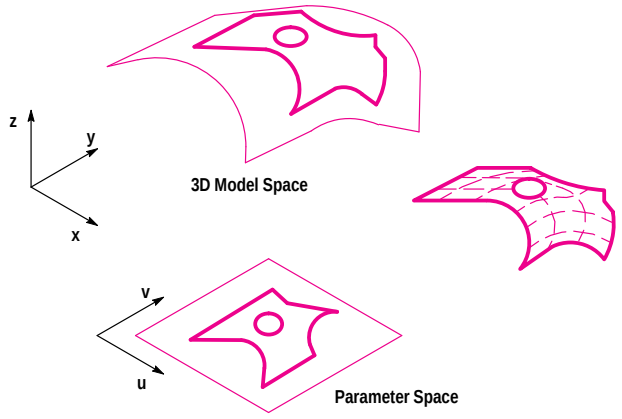


Fig. 4. Trimming in parameter space (p-space).

Table III
Model Mapping

Entity	Trimmed	Trimmed Parametric	Untrimmed	Wireframe
Parts and Assemblies	308+408	308+408	308+408	308+408
Faces	Entity 143	Entity 144	Entity 402	
Loops	Entity 141	Entity 142	Entity 102	
Edge+Base Curve	Curve Entity	Curve Entity	Curve Entity	Curve Entity
Base Surface	Surface Entity	Surface Entity	Surface Entity	None

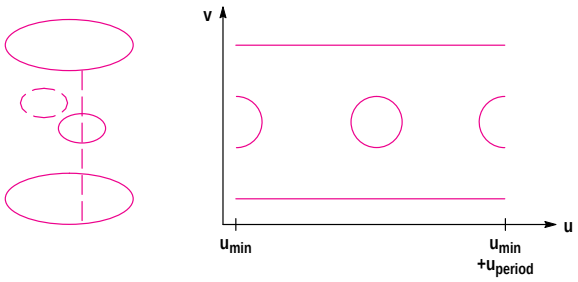


Fig. 5. Cylinder topology in 3D and p-space.

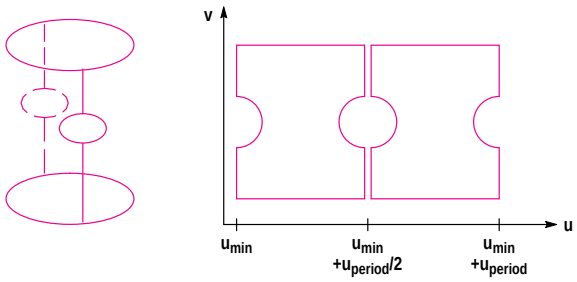


Fig. 6. Periodic surfaces in 3D and p-space after splitting.

two or four parametrically trimmed surfaces (u- and v-parametric surfaces (toruses)) in the IGES model. Fig. 6 illustrates this situation.

Another problem with parametric trimmed surfaces are surface singularities. Singular points are points where the surface derivatives and normal are not well-defined. For such points there is not always a one-to-one mapping from 2D parameter space to 3D model space. This means there is an infinite set of (u,v) points in parameter space that result in the same 3D model space point. Such singularities are easily created by rotating profiles around an axis where the profile touches the axis. Examples are cones, spheres, degenerated toruses, triangular spline patches, and so on (see Fig. 7).

HP PE/SolidDesigner is designed to handle singularities as a valid component of a model. They are marked with a vertex if they are part of a regular loop or with a special vertex loop if they are isolated from the remaining loops. However, it is not possible to express singularities in trimmed parametric surfaces legally in IGES.

To resolve this issue we reduce the singularity problem to the problem of the valid representation of triangular surfaces. The splitting algorithm just described is applied so that all singularities are part of a regular loop. Thus, we are always faced with the situation illustrated in Fig. 8.

Each singularity of a face is touched by two edges, one entering and one leaving the singular vertex. Knowing how triangular surfaces are handled in potential receiving systems, we offer four ways to export this kind of geometry. These are the four possible combinations of closed or open parameter loops and avoiding or using singularities.

Some systems do not need closed p-space loops, while others strictly expect them. If the closed option is chosen, the endings of the p-curves are simply connected with a straight line.

Geometrical algorithms usually become unstable near singularities. Some systems are not prepared to handle this situation and will fail. To avoid this, it is possible to shorten the parameter curves when entering or leaving a singular vertex and connect them at a numerically safe distance. This distance is measured in 3D space and is also configurable. It usually varies between 0.1 and 0.001. This will result in a surface where the region around the singularity is cut out. Fig. 9 illustrates the four possible singularity representations.

Wireframe Mode

For the wireframe mode HP PE/SolidDesigner also avoids the cylindrical topology, because in some cases information about shape would be lost (e.g., a full surface of revolution). After applying the face splitting algorithm all edges of the selected faces and parts are translated. No surface information is contained in the resulting IGES file.

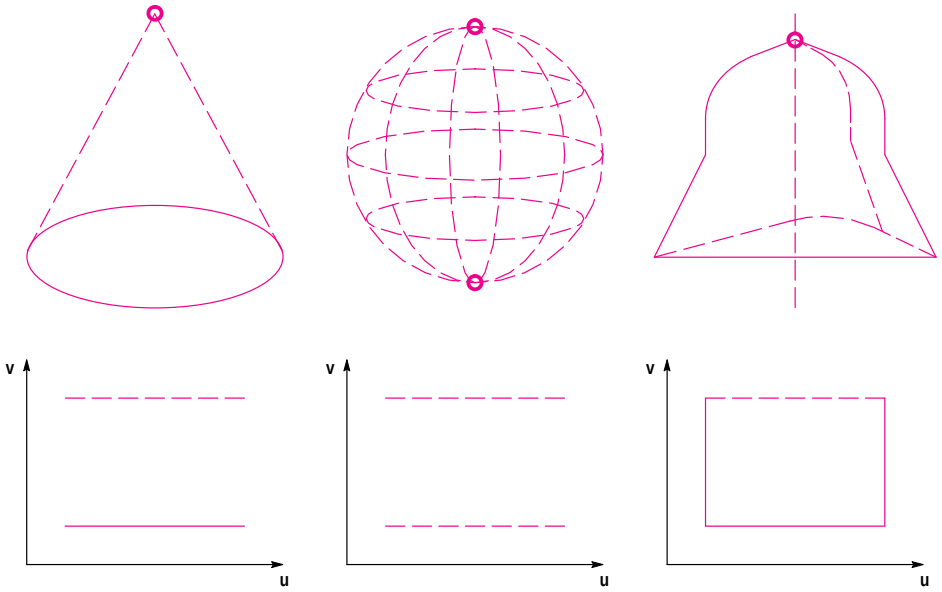


Fig. 7. Examples of surface singularities in parameter space.

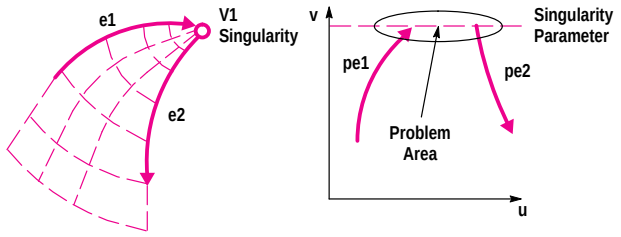


Fig. 8. Triangular surface situation.

Extracting Solid Information from Surface Models

IGES surface data from solid modelers often contains all surfaces of a closed volume or a connected face set. However, the connectivity between adjacent faces is lost. If the surface model fullfills some specific requirements it is possible for the receiving system to recompute this missing information. The following describes these requirements and shows how connectivity between faces can be reestablished. This method can be used to create a solid model from HP PE/SolidDesigner IGES output.

Automatic comparison of all boundary curves on coincidence or reverse coincidence would be a very time-consuming and numerically unstable task. However, it is common for the endpoints of the trimming curves of adjacent faces to be coincident within a very small accuracy. This makes it possible to identify trimming curves that share common start points and endpoints. If the two faces of these trimming curves have the same orientation one can try to connect the faces to a face set. For this task one must try to find a geometry for a common edge that fulfills the following accuracy constraints (see Fig. 10):

- The curve is close enough to surface 1.
- The curve is close enough to surface 2.
- The curve is close enough to curve 1.
- The curve is close enough to curve 2.

The first candidates for such a curve are the original trimming curves, curve 1 and curve 2. If either satisfies all four requirements it is incorporated into both face descriptions and the connection is established. If neither curve can be used, one can try a combination of the two, or reduce the receiving system's accuracy.

This method fails if the face orientation is inconsistent or if adjacent faces do not share common start points and endpoints.

Importing IGES Wireframe Data

IGES wireframe data can be easily imported into HP PE/SolidDesigner, since HP PE/SolidDesigner's kernel supports wire bodies. The modified wire data can be saved in HP PE/SolidDesigner's data format. Possible uses for this capability include migration from old-line systems to HP PE/SolidDesigner, interaction with different sources and suppliers, and communication with manufacturers.

In HP PE/SolidDesigner a wire is defined as a set of edges connected by common vertices. A body consisting only of wires is called a wire body. IGES 3D curve data is used to generate the edges of a wire body. This includes lines, circles, B-splines, polylines, and composite lines. IGES surface data such as trimming curves of trimmed surfaces are also used to generate edges. To simplify later solid model generation the axis and generatrix of a surface of revolution are also transformed into edges for the wire body. Since only edges have to be generated for a wire body, there are no accuracy problems as described above for IGES surface importation. On the other hand, information on B-spline surfaces is lost.

Wire data imported from an IGES file is collected into an assembly. The assembly gets the name of the IGES file. Any substructure of the IGES file like grouping in levels is transformed into parts within the assembly. Thus, hierarchical information contained in the IGES files is maintained within HP PE/SolidDesigner. The generated parts can be handled like any other part in HP PE/SolidDesigner. To distinguish

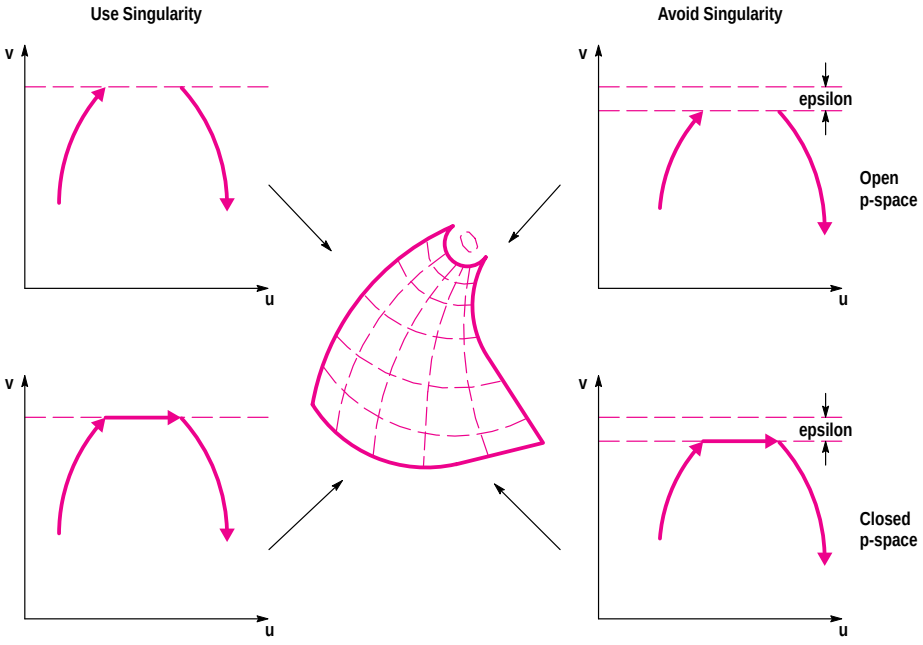


Fig. 9. Four possible singularity representations.

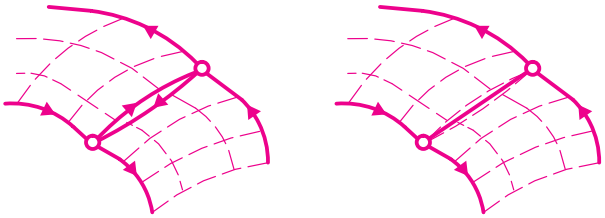


Fig. 10. Finding a common edge for adjacent faces.

wire parts, they can be colored. The options of HP PE/SolidDesigner's show menu work for the parts as well as the settings of the part container. A wire part can become the active part. The edges and vertices of a wire part are displayable, all browsers work with wire parts, and wire parts can be moved or become members of an assembly.

To build a solid model from a wire body, the edges and vertices of the wire body can be used to position a workplane. Then edges of the wire body can be selected and projected onto the workplane. The resulting profile can then be used to create a solid, for example by measuring an edge length needed for an extrude operation.

Fig. 11 shows an example of an IGES wireframe model with four parts and the resulting solid model. Automatic generation of solids from wires could be implemented but freeform surface information would probably be lost. The real benefit of wireframe import is for reference purposes.

STEP-Based Product Data Exchange

Manufacturing industries use a variety of national and industrial standards for product data exchange. These include IGES for drawing and surface exchange (international), VDA-FS for surface exchange (mainly the European automotive industry), and SET for drawing and surface exchange (France and the European Airbus industry). This variety of different incompatible standards causes a lot of rework and waste of valuable product development time which cannot be afforded if companies are to survive in the competitive marketplaces of tomorrow. Today's standards, originated in the early 1980s, are no longer satisfactory for product data description and exchange. Standards like IGES or VDA-FS, which are limited to surface or engineering drawing exchange, do not adequately handle other explicit product data categories such as product structure or assemblies or geometric solid models.

Industry trends today are characterized by internationalization of manufacturing plants which are spread over the continents of the globe, and by lean production in which many parts are subcontracted or bought from local or international suppliers. National standards and incompatibilities between existing standards are obstacles to these trends and will have to be replaced by international standards.

Large companies in the aerospace and automotive industries in the U.S.A. and Europe have now taken the offensive towards the implementation and use of STEP (*St*andard for the *E*xchange of *P*roduct Model Data) as an international standard for product data exchange and access, starting in 1994. Companies such as BMW, Boeing, Bosch, General Motors, General Electric, Daimler-Benz, Pratt&Whitney, Rolls Royce, Siemens, and Volkswagen have been using STEP prototype implementations in pilot projects with promising results.

Ultimately, STEP is expected to meet the following requirements for an international product data exchange standard:

- Provides computer interpretable and standardized neutral product model data. Neutral implies compatibility with any CAD or CIM system that best fits the design or manufacturing task.
- Implements the master model concept for product data. The entire set of product data for a product with many single parts is kept in one logical master model which makes it possible to regenerate the product as a whole at a new manufacturing site. This means that product assemblies, including administrative data and bills of material, are handled.
- Provides completeness, conciseness, and consistency. This requires special data checking and validation mechanisms.
- Provides exchangeable product data without loss. The product data must be exchangeable from one CAD or CIM system to another without loss of data.

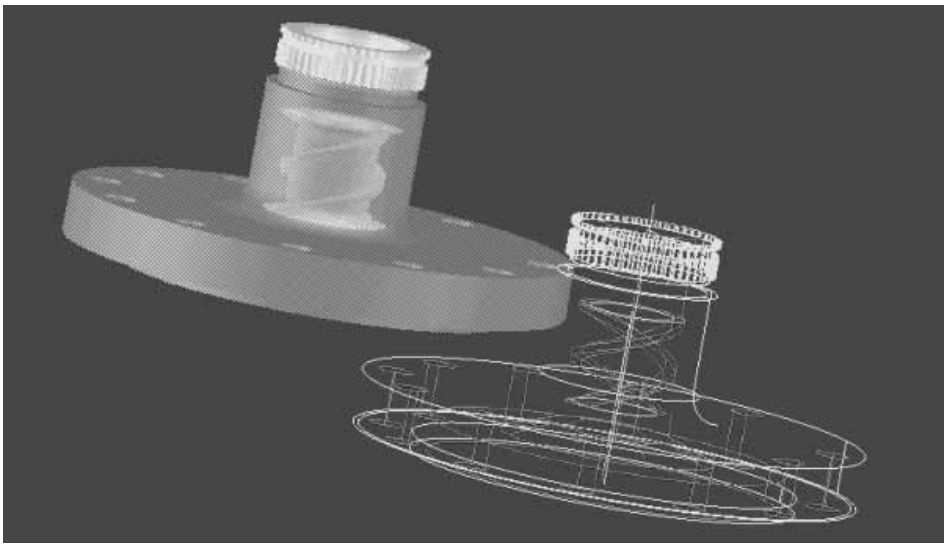


Fig. 11. Imported wire-body and the solid model constructed by HP PE/SolidDesigner.

- Provides long-term neutral data storage and interpretability. Product data is an important asset of a manufacturing company. The product data should be retrievable and interpretable by any CAD or CIM system after a long period of time, say 10 years or more. This is a significant challenge.

These requirements cannot be satisfied immediately. The STEP program also has shorter-term priorities for standardizing specific subsets of the product data. These include:

- The complete 3D geometric shape in the form of a 3D boundary representation solid model (B-Rep solids)
- Surface model and wireframe model data
- Product structure and configuration data.

Another priority is product documentation. An important goal is consistency of the engineering drawing with the 3D product geometry.

STEP Overview

STEP, the Standard for the Exchange of Product Model Data, is the ISO 10303 standard. It covers all product data categories that are relevant for the product life cycle in industrial use. STEP describes product data in a computer interpretable data description language called *Express*. The STEP standard is organized in logically distinct sections and is grouped into separate parts numbered 10303-xxx (see Fig. 12).

The resource parts of the standard describe the fundamental data and product categories and are grouped in the 1x, 2x, 3x, and 4x series. The Express data description language is defined in part 11. All other product description parts use the Express language to specify the product data characteristics in the form of entities and attributes. In addition to the product description parts there are implementation resources which are given in part 21, the STEP product data encoding scheme (the STEP file), and part 22, the Standard Data Access Interface (SDAI), which provides a procedural method for accessing the product data. There are different language bindings for part 22, such as C or C++ programming languages. The 3x series parts specify conformance requirements for STEP implementations.

Examples of STEP-standard resource parts are the fundamentals of product description and support (part 41), the geometrical shape (part 42), the product structure (part 44), material (part 45), the product presentation (part 46), tolerances (part 47), and form features (part 48). The application-specific resources are grouped in the 1xx series. Examples are drafting resources (part 101), electrical (part 103), finite element analysis (part 104), and kinematics (part 105). On top of the resource parts and application resources are the

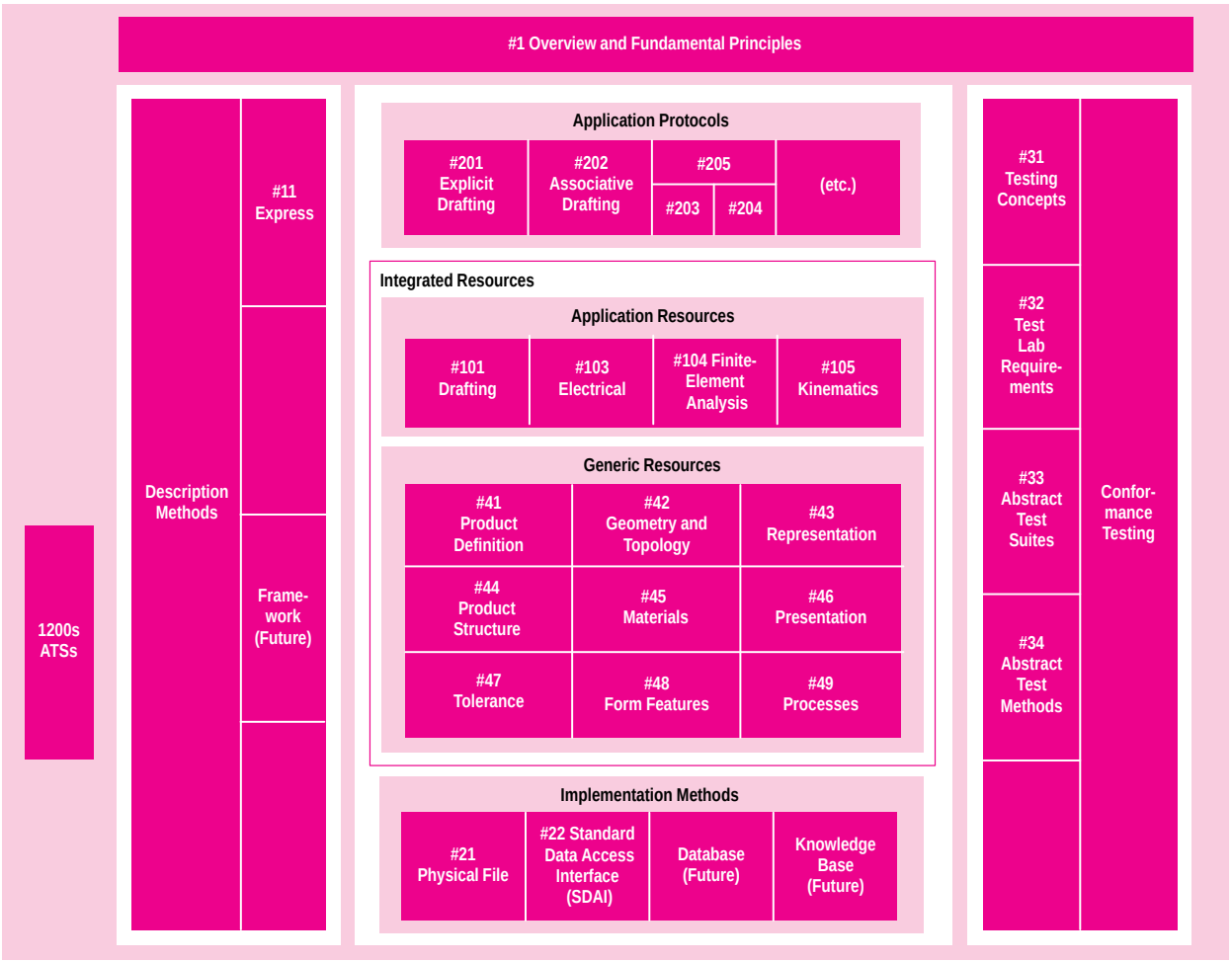


Fig. 12. Architecture of ISO 10303, Standard for the Exchange of Product Model Data (STEP).

application protocols (AP) which use the underlying resources in a specific application context, such as mechanical design for discrete part manufacturing, and interpret the resource entities in the application-specific context. STEP implementations for CAD or other computer-aided systems are based on application protocols. Application protocols are under definition for application areas like basic drafting, associative drafting, mechanical design, electrical design, shipbuilding, piping, architecture, and others. Here, we highlight just two examples, AP203 and AP214.

AP203: Configuration-Controlled 3D Design. AP203 was developed under the leadership of PDES Inc. It covers the major requirements for U.S.-based industries such as the aerospace industry for government and industrial manufacturing contracts. The product data covered in AP203 includes geometric shape (B-Rep solid models, surface models, wireframe models), product structure, and configuration management. AP203 is the underlying STEP specification for many CAD and CIM system implementations.

AP214: Core Data for Automotive Mechanical Design. AP214 has been developed by the automotive industry and covers product data categories relevant for the design and manufacturing of automotive parts and products. AP214, initiated in Germany and internationally supported, is still under finalization in parallel with its industrial implementation in CAD and CIM systems. The implementations have been coordinated and harmonized in the European ProSTEP consortium and the implementation is focused initially on the geometrical product descriptions (solid models, surface models) and product structure. However, all other kinds of product data categories relevant for mechanical design in the automotive industry (e.g., form features, materials, tolerances) are within the scope of AP214 and are going through the standardization process.

Initial Release

The initial release of STEP parts focuses on the most urgently needed kernel definitions of the standard, which cover the geometrical shape description, including all topological information, the product structure, and the configuration management data. Basic product documentation in the form of low-level engineering drawings is also covered. The parts included in the initial release are parts 1, 11, 21, 31, 41, 42, 43, 44, 46, 101, 201, and 203. The first two application protocols to become standards are AP201: Explicit Drafting and AP203: Configuration-Controlled 3D Design.

Upcoming releases of STEP will cover the next priorities in the area of drafting, such as AP202: Associative Drafting, materials, tolerances, form features, and parametrics, and other application protocols such as AP204: Mechanical Design Using B-Rep Solid Models and AP214: Core Data for Automotive Mechanical Design.

HP Involvement in STEP

HP has been working on the standardization of product model data since 1989 and has focused on the emerging international standard STEP for 3D product data. The product data focus has been on 3D kernel design data, completeness of topology and geometry, B-Rep solid models, and product structure and assemblies, as well as on associative drafting documentation. HP is an active member in organizations that have an impact on the ISO STEP standard, and contributes to STEP through national standards organizations in the U.S.A. (e.g., NIST, ANSI) and Europe (e.g., DIN in Germany). Of particular interest are the organizations PDES Inc., PRODEX, and ProSTEP.

PDES Inc. HP has concentrated on three major areas of PDES Inc.'s STEP activities: mechanical design of 3D product data, associative drafting for CAD data, and electronic data definition and exchange.

The mechanical design initiative of the U.S. aerospace and aircraft industries, the automotive industry, and the computer industry resulted in STEP application protocol 203. HP, a PDES Inc. member in the U.S.A. and an ESPRIT CADEX member in Europe, contributed to the 3D geometric design definition of AP203 in a joint effort of PDES Inc. and CADEX. The AP203 3D geometries cover solid models, surface models, and wireframe models and are shared by other application protocols, thereby promoting interoperability between different application areas.

HP has also been actively supporting the U.S. initiative to define a good-quality standard for associative drafting documentation in STEP. Associative drafting, covered by AP202, is considered an integral portion of the product data for contractual, archival, and manufacturing reasons. For example, government contracts and ISO 9000 require that product data be thoroughly documented. This includes engineering drawing data of a product in addition to the 3D product data and the configuration data. Electronic design and printed circuit board design data are also covered in STEP.

PRODEX. In 1992 participants in the ESPRIT CADEX project demonstrated publicly the first B-Rep solid model transfer via STEP for mechanical parts in Europe. To develop this new technology the PRODEX project was founded in 1992 with the goal of developing STEP data exchange for CAD design, finite element analysis, and robot simulation systems. Twelve European companies joined the project. So far, the project's achievements include the definition of a STEP implementation architecture, the development of a STEP toolkit, and the development of STEP preprocessors and postprocessors.

Product data exchange between the different vendors is ongoing and shows very promising results for CAD-to-CAD data exchange, CAD-to-finite-element-system exchange, and CAD-to-robot-simulation-system exchange. The STEP standard has been further fostered by a joint effort with the ProSTEP project to develop AP214, in cooperation with the U.S., European, and Japanese automotive industries.

ProSTEP. ProSTEP is an automotive industry initiative for a highway-like STEP product model data exchange. In 1992 the German companies Bosch, BMW, Mercedes-Benz, Opel (GM), Volkswagen, and Siemens launched an initiative to bring the major CAD vendors together with the goal of implementing the first harmonized set of STEP product data exchange processors (product data translators) for industrial use in the automotive industry. The approach taken was to compile the user requirements, to build on the results and experiences of the ESPRIT CADEX project, and to launch at the ISO level a STEP application protocol, AP214, which covers the core data for automotive mechanical design.

The following CAD/CIM systems are involved in the project and have STEP data exchange processors either available or under development: Alias, AutoCAD, CADDS/CV-Core, CATIA, EUCLID3, HP PE/SolidDesigner, EMS-Power Pack, I-DEAS Master Series, SIGRAPH STEPIntegrator, SYRKO, Tebis, ROBCAD, and others.

The initial focus in ProSTEP for STEP products is on design data exchange for 3D geometry: B-Rep solid models, surface models, and wireframe models. For migration from legacy systems, wireframe data needs to be supported, at least for data import. Communication with applications like numerical control (NC) programming systems today typically requires surface model data, although in the future more solid model data will be used. Initially, the HP emphasis is on bidirectional product model exchange (input and output) of 3D B-Rep and surface models.

STEP Tools Architecture

In STEP implementation projects, standardization has been extended beyond the product data to the STEP implementation tools. The CADEX, PDES Inc, PRODEX, and ProSTEP projects have all taken this approach.

A standardized STEP tool architecture provides the following benefits. These include shareability of tools between different implementors, shortened development time for STEP processor implementations (software development productivity gain), increased likelihood of compatibility between STEP implementations (differences in STEP definition interpretations are minimized), parallel development of tools (concurrent engineering), extendability of tools to track new standardization trends, increased flexibility (new STEP models require fewer code changes), and centralized maintenance of tools.

Fig. 13 shows the PRODEX STEP tools architecture. The functional blocks of a STEP toolkit or STEP development set are:

- STEP Standard Data Access Interface (SDAI),
- STEP Express compiler
- STEP file scanner/parser
- STEP file formatter
- STEP data checker
- STEP conversion tool.

The main interface to the STEP application is the STEP Standard Data Access Interface, which provides a computer programming language for dynamic access to the STEP data. Application-specific mapping and conversions are implemented on top of this interface.

The Express compiler conveys the product data descriptions contained in an Express schema (the metadata of the data model) to the toolkit. It contains an Express file reader and compiles the file contents to the internal representation of the data model. The SDAI is the recipient of the product data metamodel and uses the metamodel as a reference for the product instance data, which is imported through the STEP file scanner/parser.

The STEP file scanner/parser reads (scans and parses) the STEP instance data contained in a STEP data file and uses the currently valid metamodel for checking the syntax of the imported instance data.

The STEP file formatter formats the data to a part-21-conformant STEP file which is read from the SDAI by using the current valid metadata (e.g., a specific application protocol such as AP203).

The STEP data checker is a validation tool that checks the instance data currently in the SDAI based on the corresponding metadata model, which is also contained in the SDAI. The checking covers consistency checks like references between entities (e.g., existence dependency), and rule checking, which is covered in the metamodel. The checking is optionally applicable to the data in the SDAI. It is very helpful during the development of processors, for checking new metadata models, or for checking the first data imported from a new system.

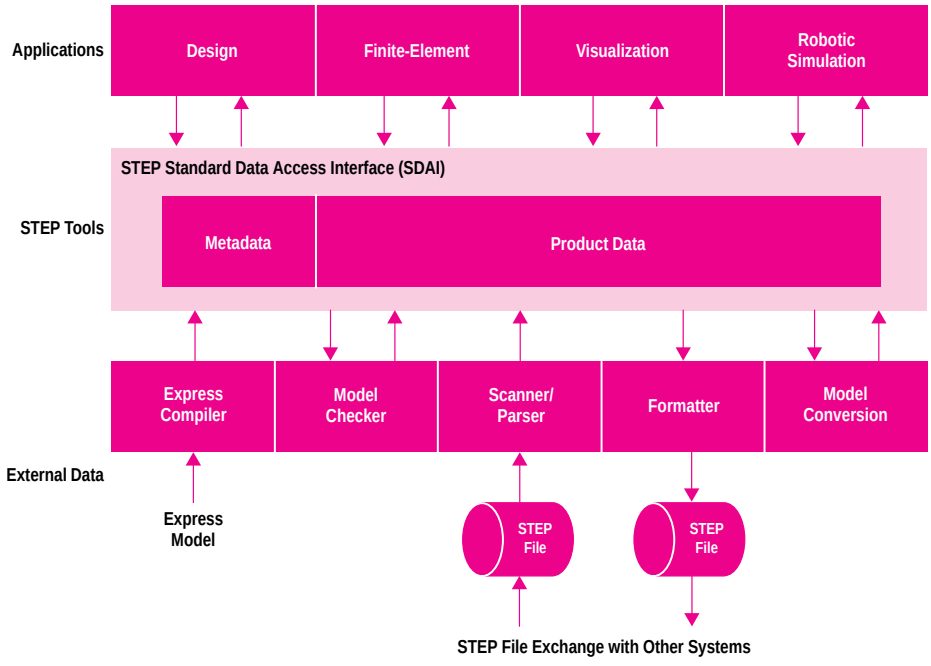


Fig. 13. PRODEX STEP tools architecture.

The STEP conversion tool is a pool of conversion functions (a library) that includes all kinds of geometrical, topological, and other model conversions. The focus is on geometrical conversions which are heavily used for data exchange between systems with different geometric modeling concepts. For example, one CAD system might use rational polynomial representations for its inherent geometric representation of curves and surfaces (e.g., NURBS, nonuniform rational B-splines), while the other might use nonrational representations (e.g., NUBS). In this case an approximation to the nonrational representation has to be applied, at the price of increasing the amount of data. For another example, a surface modeling system might export trimmed surface data with curve representations in 2D parameter space, whereas the receiving system might handle only 3D space curves. In this case the 2D parameter curves have to be evaluated and converted to 3D trimming curves in 3D space.

By using a STEP toolkit the requirements for the implementation of a STEP processor might be reduced to just the native data interface to the STEP tools, which consists of the data output to the SDAI (for the STEP preprocessor) and the data imported from the SDAI (for the STEP postprocessor).

The main task in linking a CAD system to the toolkit consists of defining and implementing the mapping between the system internal representation and the standardized entity representation in the schema of the standard (e.g., an application protocol).

HP PE/SolidDesigner STEP Implementation

The target application protocols for HP PE/SolidDesigner are initially AP203 and AP214, in which both solid and surface models are supported. In addition to the HP PE/SolidDesigner internal data models, the solid and surface models of other CAD systems are of major interest. With the introduction of STEP, B-Rep solid model data exchange comes into industrial use, representing a new technology shift. HP PE/SolidDesigner has its focus on solid models and is best suited for STEP-based bidirectional solid model exchange. However, surface models are also supported.

In addition to the geometric specifications, product information and configuration are covered in the implementation. In this article, the geometric and topological mappings are discussed. The assembly, product structure, and administration mappings are not covered.

STEP Preprocessor (STEP Output)

The preprocessor exports the HP PE/SolidDesigner model data in a STEP file. The preprocessor takes care of the mapping of the HP PE/SolidDesigner model to the STEP model.

The internal geometrical and topological model of HP PE/SolidDesigner is in many respects similar to the STEP resources of part 42 of the STEP standard. Hence the mapping is often straightforward. On the other hand, there are data structure elements that are not mapped to the STEP model.

HP PE/SolidDesigner uses the following geometric 3D elements:

- Analytics: 3D surfaces such as planes, cones, cylinders, spheres, and toruses, and 3D curves such as lines, arcs, circles, and B-splines
- Nonanalytics: typically 3D elements such as B-spline curves and surfaces, and linear and rotational swept surfaces.

The topology used for the exchange of solid models is based on the manifold topology of STEP part 42. The elements used are manifold solid boundary representations, closed shells, faces, loops, edges, and vertices. The link between the topology and the geometry is given by references from faces to surfaces and from edges to curves. The geometrical points are referenced by vertices.

The HP PE/SolidDesigner STEP surface models are also based on topological representations. Special elements are used for surface models, such as shell-based surface models and closed and open shells. The other underlying topolocial elements are the same as in the solid models. The geometric representations of the surfaces are typically the same as in the solid model representations.

STEP Postprocessor (STEP Input)

The HP PE/SolidDesigner postprocessor supports the import of B-Rep solid models and surface models along with the necessary product structure data. The postprocessor is

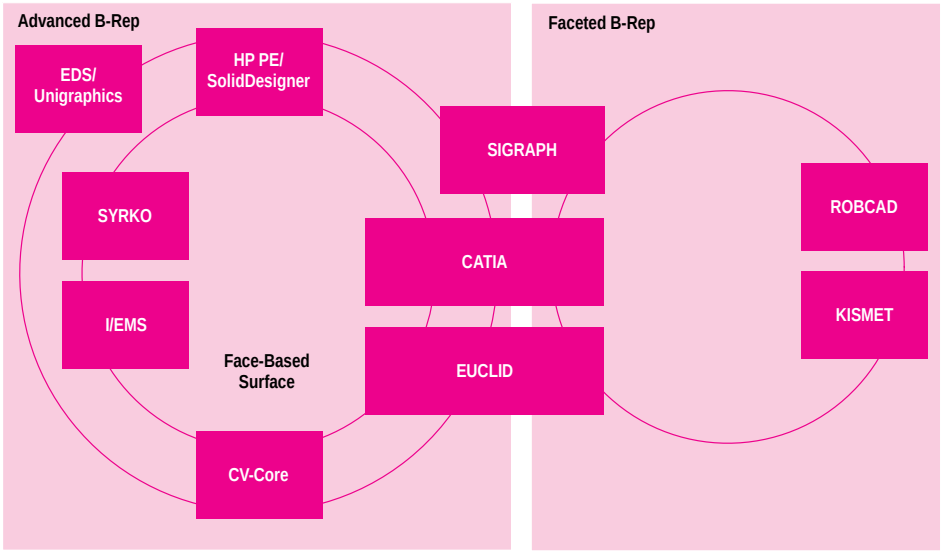


Fig. 14. Data exchange cycles between different CAD systems, including robot simulation systems, in the ProSTEP project.

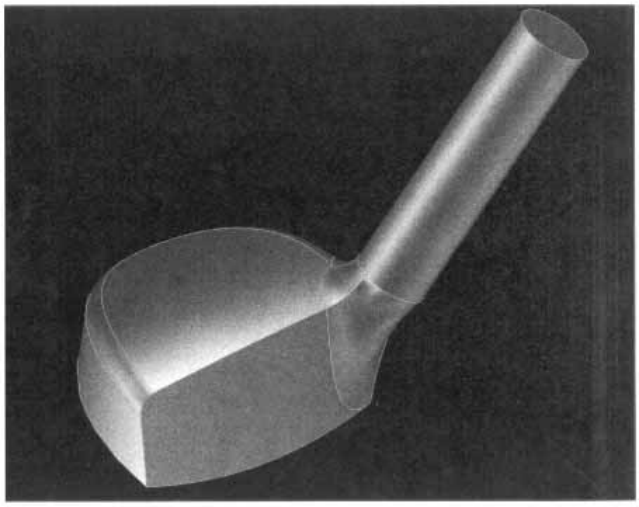


Fig. 15. Golf club solid B-Rep model imported into HP PE/SolidDesigner from CATIA (CAP-Debis).

capable of covering at least the functionality of the preprocessor so that it is possible to store and retrieve HP PE/SolidDesigner data in a STEP file representation (this is called the *short cycle test*).

The STEP postprocessor imports STEP files from other systems based on specifically supported application protocols. Postprocessing is one of the most difficult tasks in data exchange, especially when the data imported comes from a system that is very different from the receiving system. Potential problems arise in postprocessing if the sending and receiving systems have different accuracies, use different modeling techniques to generate the data, have different or missing surface connectivity, use different algorithms or criteria to determine surface intersections or connectivity, or use different model representations for similar model characteristics.

When surface models are imported, it cannot be guaranteed that they can be migrated to solid models even with user interaction. However, in special cases imported surface models can be migrated to solid models without problems. In many cases imported surfaces provide boundary conditions for the solid model. In most cases the data can be used as reference geometry to check interference or provide dimensions for the solid models. For example, an imported surface set might represent the surrounding boundary geometry within which the final mechanical part has to fit without interference.

Importing surface models into HP PE/SolidDesigner is considered important and critical since many other CAD systems, especially legacy systems, often support only surfaces or wireframe models, not solid models. Therefore, the postprocessing of STEP surface models needs to cover a broader scope than the preprocessing. Sometimes, different surface representations are used in different application protocols, such as AP203 and AP214. Hence, different external representations may need to be mapped to one internal representation in HP PE/SolidDesigner.

In the initial implementation of the HP PE/SolidDesigner postprocessor, topology bounded surface models are supported. These provide the most sophisticated description of the connectivity of the individual surfaces used in a solid model. Geometrically bounded surface models are supported as a second priority.

The Accuracy Problem

When importing CAD data from other systems the accuracy of the data plays a key role and determines whether a coherent and consistent CAD model can be regenerated to represent the same kind of model in the receiving system.

Let's define the term accuracy. There are different accuracy or resolution values that must be considered in geometric modeling and CAD systems. For 3D space, a minimum linear distance value (a length resolution value) can be defined, which is the absolute distance between two geometric points that are considered to coincide in the CAD internal algorithms; this represents the zero distance. We'll call this value

* Often, CAD surface models are not consistent because the generating system lacks checking mechanisms or does not track connectivity. Very often, consistency and accuracy are the responsibilities of the user of the system rather than under system control.

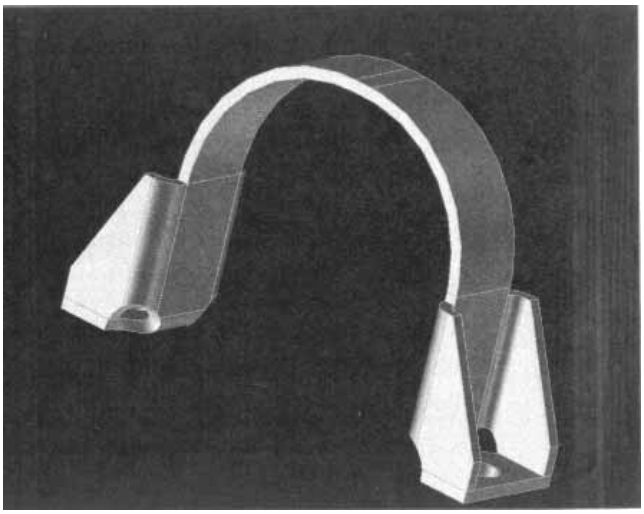


Fig. 16. Clamp solid B-Rep model imported from Unigraphics II (EDS).

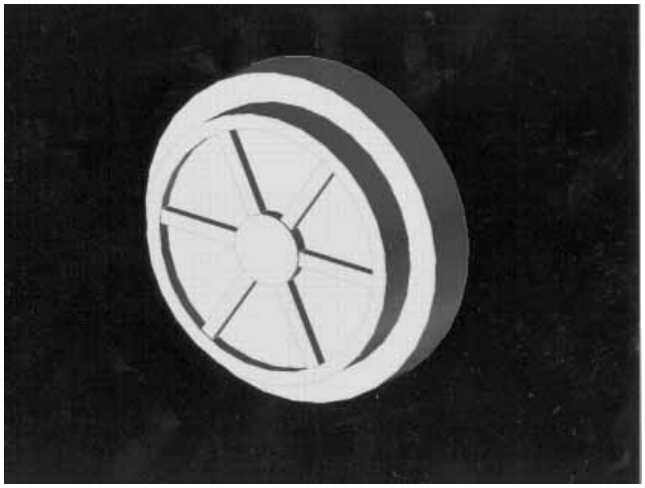


Fig. 17. Wheel solid model imported from SIGRAPH-3D (Siemens-Nixdorf).

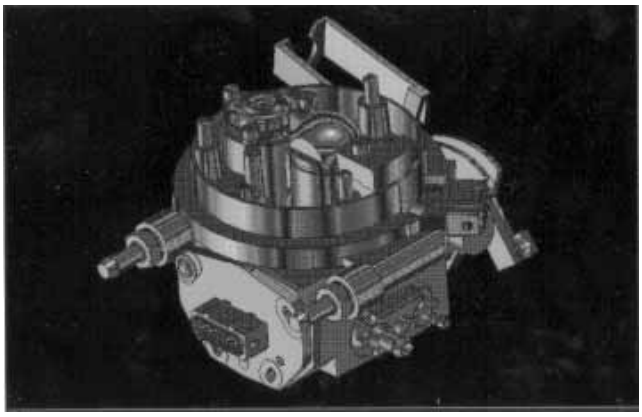


Fig. 18. B-Rep model imported from Unigraphics (EDS).

the *linear accuracy*. A typical value could be 10^{-6} mm which is highly accurate for many mechanical design applications. A similar value can be specified for *angular accuracy*, *parametric accuracy*, and so on. The discussion here is limited to linear accuracy.

If the sending system uses a higher linear accuracy (more precise data) than the receiving system, distinct geometric points will be detected to coincide in the receiving system. This might result in a change in the topology (which might cause further inconsistencies) or the geometry. If the sending system uses a lower linear accuracy (less precise data), the receiving system might complain that the topology is not correct or the geometry and the topology are inconsistent.

To prevent or at least minimize these kinds of accuracy problems it should be possible to adjust the accuracy in the receiving system to the accuracy values of the data to be imported. For example, if the sending system uses a different accuracy for the model generation process, say a linear accuracy of 10^{-2} mm, then the receiving system should adjust its internal algorithms to the same accuracy.

Experience with HP PE/SolidDesigner has shown that this kind of adjustable accuracy helps regenerate CAD models that were generated in different systems with different accuracies. Also, for data models composed of components with different accuracies, the components can be brought together on the assembly level to form a complete product model.

In the STEP implementation of AP214 an adjustable linear accuracy value is conveyed in the STEP file to tell the receiving system the appropriate accuracy value for postprocessing.

User Features

The user can select via the HP PE/SolidDesigner graphical user interface the objects (e.g., several B-Rep bodies) to put into a STEP file. For example, the user decides whether to send the data in a B-Rep solid model or a surface model representation. The user can choose some configuration parameters that help tailor the model data set for best communication to a specific target application. However, all data must comply with the STEP standard.

When importing (postprocessing) a STEP file the user can define some parameters that ease the processing of data. For example, the user might set the accuracy value before importing a data set that was designed with a specified accuracy, or might choose to convert the imported data to a different representation.

STEP Model Exchange Trials

Various STEP file exchanges have been performed within the last 12 months, not always with satisfying results. This has resulted in more development work by the exchange partners. This process of harmonizing the STEP preprocessors and postprocessors of different CAD vendors is considered to be of vital importance for the acceptance of the STEP standard and its application protocols. Within the ProSTEP project this process has worked particularly well. Other work has been done with, for example, AP203 implementors together with PDES Inc.

At this time, solid model data exchange can be said to be working very well, especially compared with what was possible with existing standards. STEP-based surface model exchange has also reached a level that was not possible with existing standards like IGES or VDA-FS, especially with respect to topological coherence, which is easily conveyed with STEP between many CAD systems. Of course, the wide variety of surface models, with the resulting accuracy and connectivity problems, will need to be addressed by the different CAD system vendors to optimize data transfer via STEP. In the meantime, STEP file exchange has matured to the point where STEP products are offered by various CAD vendors and system integrators.

Within the ProSTEP project one of the broadest ranges of STEP-based data exchange trials have been performed between HP PE/SolidDesigner and other CAD systems (see Fig. 14). Solid model industrial part data has been exchanged, for example, with CATIA (CAP Debis and Dassault/IBM), Unigraphics II (EDS), SIGRAPH Design and STEP Viewer (Siemens-Nixdorf), and others. Some of the successful results are shown in Figs. 15, 16, 17, and 18. Surface model industrial part data has been exchanged with CATIA, EUCLID, SYRKO (Mercedes-Benz corporate design system), and others. Some of the successful results are shown in Figs. 19 and 20.

Next STEPs

Future releases of the STEP standard covering product data categories such as materials, tolerances, form features, manufacturing process data, and others are expected in the next few months. The expected release of AP202, associative drafting, will allow documentation of the product data in engineering drawings. Work is ongoing towards the parameterization of product features, which needs further development in the STEP standard.

The expected finalization of AP214 will make it possible to convey the product data categories in STEP files and will help to reduce design and manufacturing development cycles for simple as well as complex products. This process will be supported by further extensive use of data communication networks in the various countries. The migration from existing standards is aided by several product offerings of IGES-to-STEP and VDA-FS-to-STEP data converters.

The STEP implementation technology based on the STEP Standard Data Access Interface will be broadened and used

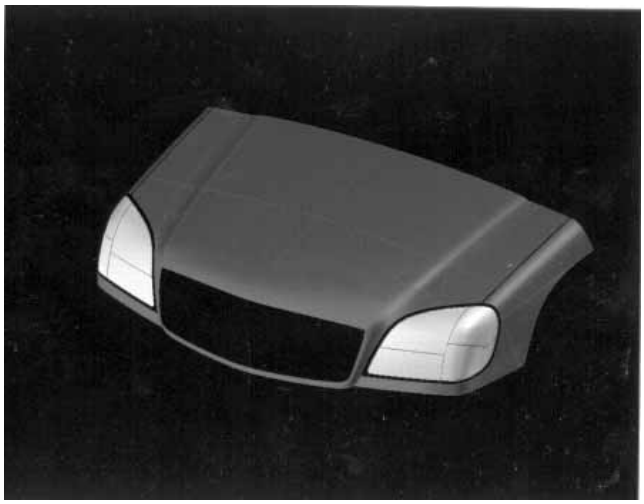


Fig. 19. Surface model imported from SYRKO (Mercedes-Benz corporate design system).

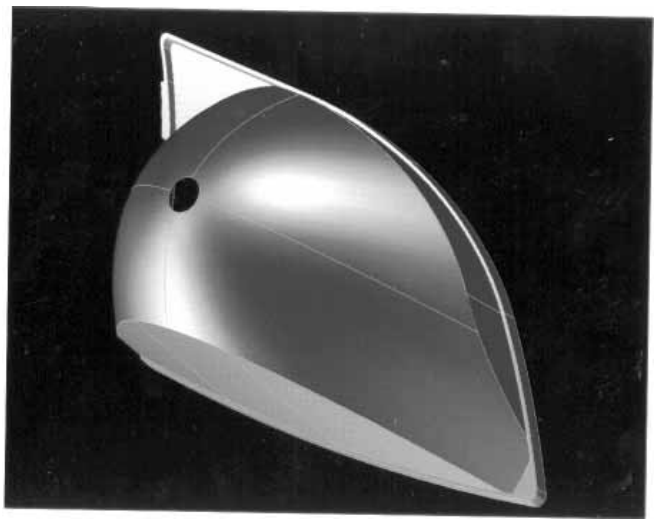


Fig. 20. Headlight reflector surface model imported from EUCLID (Matra Datavision).

in database access implementations to allow concurrent access by product design and manufacturing development.

However, for industrial use, the database technology and the STEP data access technology need to be extended and integrated. This process is expected to take several years.

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

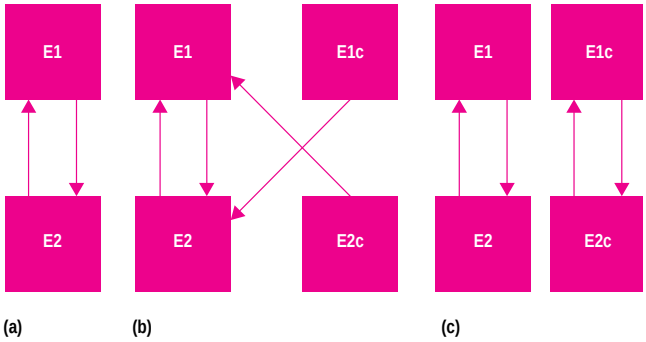


Fig. 1. Steps in copying two entities that have pointers to each other. (a) Before copying. (b) After copying. (c) After pointer conversion.

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.

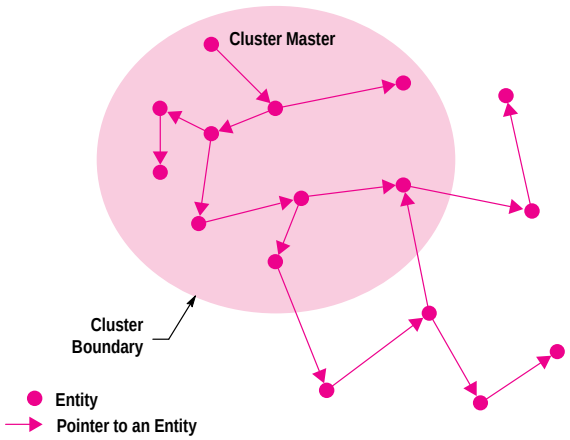


Fig. 2. A correct cluster.

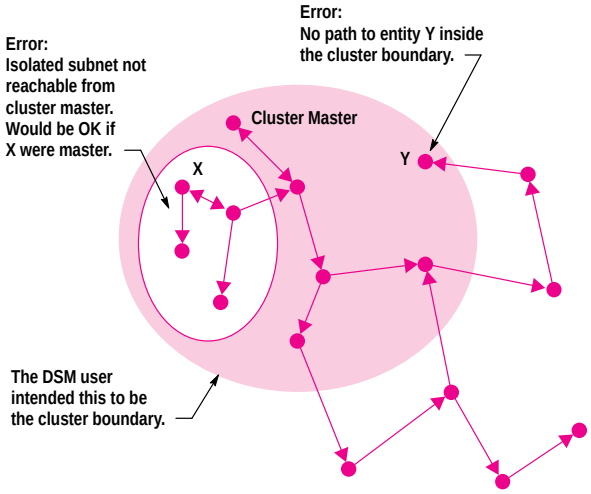


Fig. 3. An illegal cluster.

- 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.

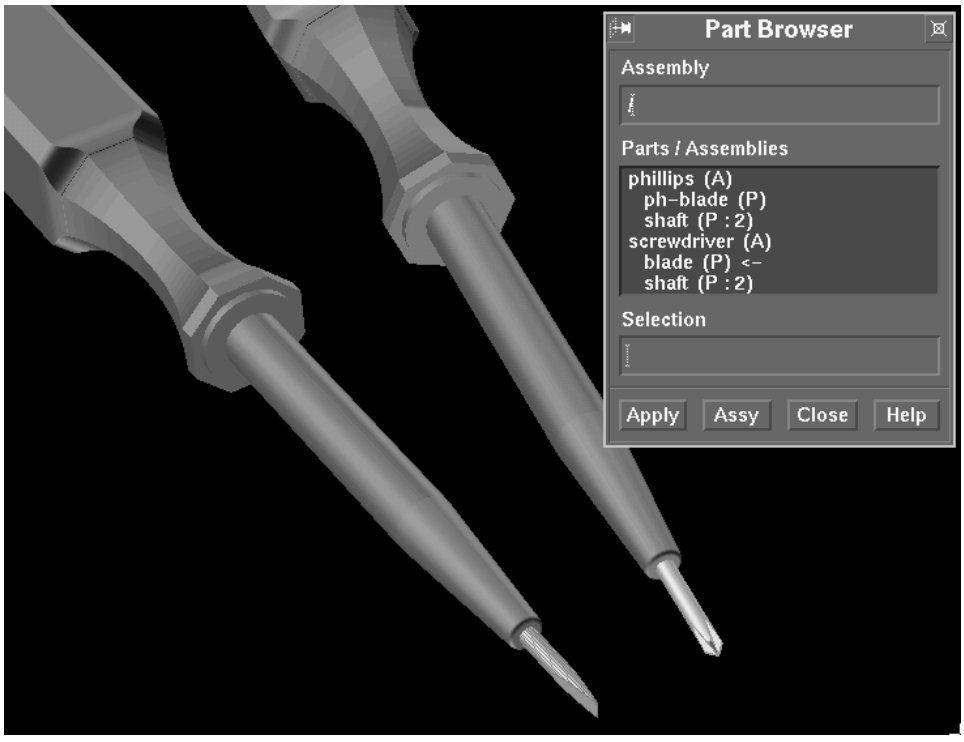


Fig. 4. Two assemblies with shared parts.

- 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

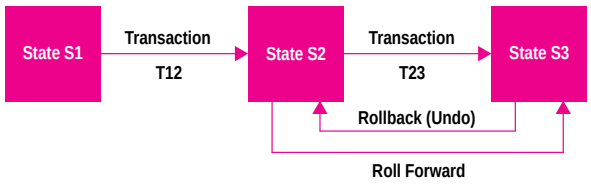


Fig. 5. HP PE/SolidDesigner transaction model. A transaction transforms one state into another. A transaction can be rolled back or rolled forward.

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.

Freeform Surface Modeling

There are two methods for creating freeform surfaces in HP PE/SolidDesigner: blending and lofting. This article describes the basics of lofting. The geometry engine, which implements the lofting functionality, uses a single-data-type implementation for its geometric interface, but takes a multiple-data-type, hybrid approach internally.

by Michael Metzger and Sabine Eismann

HP PE/SolidDesigner's kernel functionality consists of several modules that communicate through well-defined interfaces, supported by logical class definitions and hierarchies. In Fig. 1, for example, the geometric data interface for the topology engine (the Boolean engine, see article, page 74) consists of three basic elements (points, curves, and surfaces) and the corresponding utility functions like intersections. This technique makes it easy to add new functionality. For example, introducing new geometry data types is just a matter of delivering all member functions of the geometric interface for the new geometry type.

The implementation of such a concept looks simple, but reality has shown that it takes a lot of effort to keep the interface clean and to avoid copying and converting data. This is especially true for data having connections on both sides of the interface, such as pieces of a curve or curves on a surface.

The Geometry Engine

In designing a completely new implementation of the geometric kernel for a solid modeler one has a chance to avoid the problems of older implementations. What are the real problems of existing implementations? There are two fundamental approaches: NURBS libraries and hybrid methods.

NURBS libraries have only one data type: NURBS, or non-uniform rational B-splines. This data type can represent all analytics (like planes, cylinders, spheres, etc.) exactly. This means that complex freeform surfaces as well as simple analytics are represented with one single data structure. The geometrical problems only have to be solved for this single type. This sounds promising, but it turns out that the algorithmic stability does not satisfy the requirements of HP

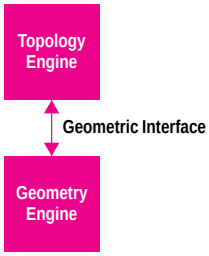


Fig. 1. The HP PE/SolidDesigner topology and geometry engines communicate through a well-defined geometric interface.

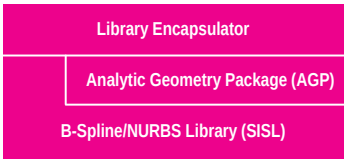
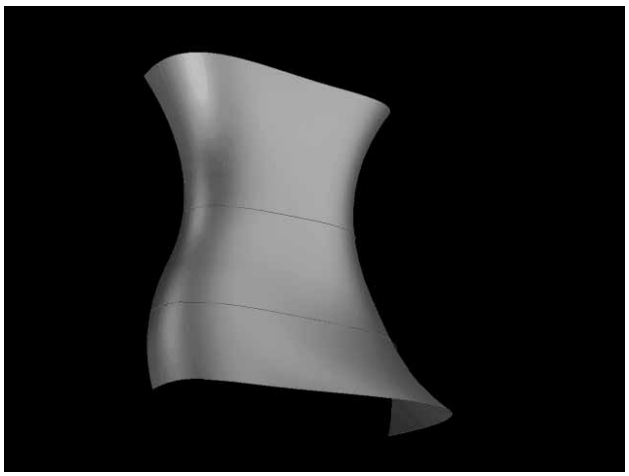
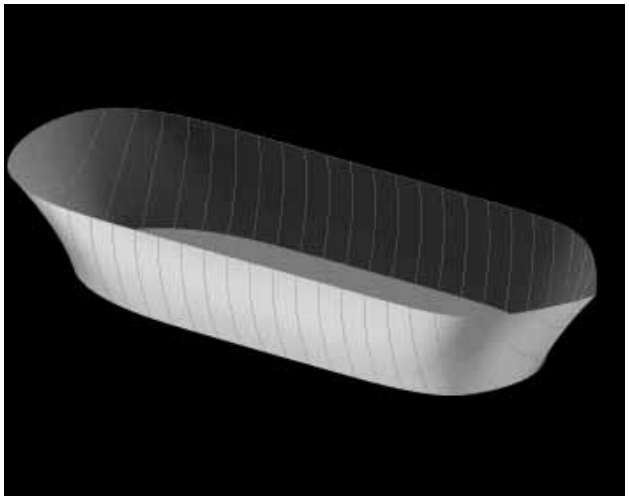


Fig. 2. HP PE/SolidDesigner geometry engine.

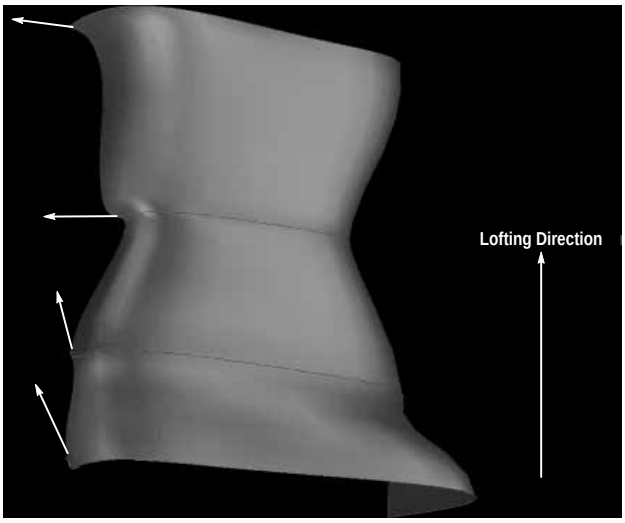


(a)

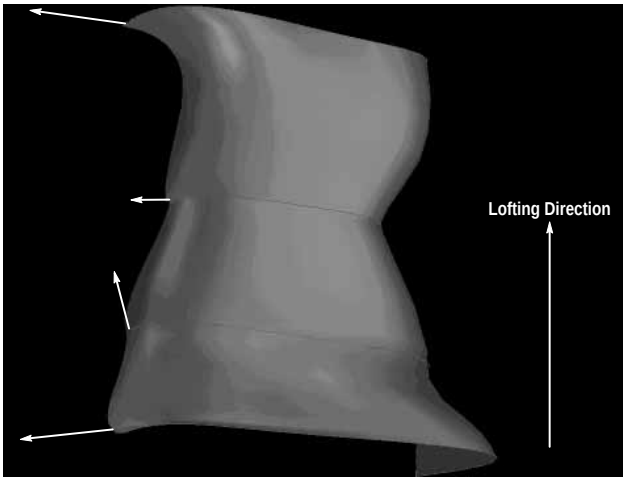


(b)

Fig. 3. (a) A lofted surface. (b) Lofting originated in ship design.



(a)



(b)

Fig. 4. Specifying the tangent profile, a kind of vector field along a curve, influences the shape of a surface.

PE/SolidDesigner. In addition, the performance is poor, especially when analytic surfaces (represented as NURBS) are intersected.

Hybrid methods are used in the HP PE/ME30 kernel (Romulus). All possible geometry data types are available, and clever special case handling results in high performance. The disadvantage is that the introduction of a new data type is an enormous effort. In addition, the Romulus kernel doesn't distinguish cleanly enough between geometry and topology, so building new functionality on this kernel can be very cumbersome and error-prone.

In HP PE/SolidDesigner we tried to combine the advantages of both approaches. The advantage of a NURBS library (one data type) is realized in the class hierarchy of HP PE/SolidDesigner: the geometric interface knows only points, curves, and surfaces. For the internal geometry structure the hybrid method was chosen. Data types include analytic types (plane, sphere, cylinder, cone, torus), semianalytic types (parallel swept B-spline, spun B-spline), B-splines, and NURBS as an extension of B-splines.

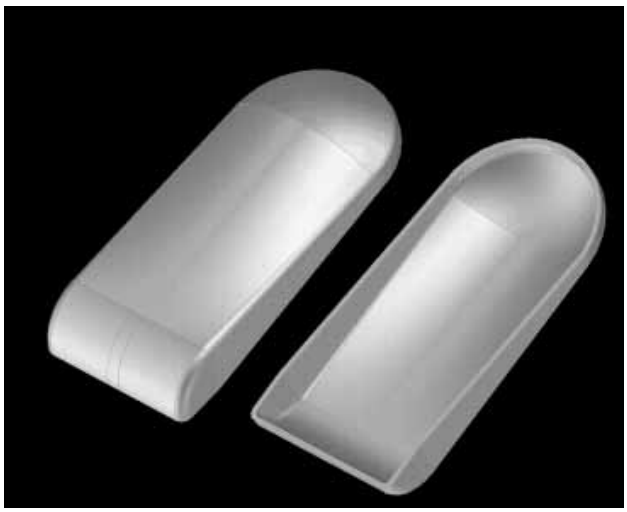


Fig. 5. Multiply connected curves.

As shown in Fig. 2, HP PE/SolidDesigner's geometry engine consists of three parts: the library encapsulator, the analytic geometry package (AGP), and the B-spline/NURBS library (SISL).

The library encapsulator delivers many convenience functions for the geometric interface and ensures its integrity. All functions dealing with geometry have to pass through the geometric interface. The only exception is a small part of the blending algorithm, which for performance reasons bypasses the library encapsulator and calls SISL directly.

The AGP was developed by DCUBED Ltd. of Cambridge, England and SISL was developed by the Senter for Industrieforskning of Oslo, Norway.

Freeform Surface Modeling

There are two methods for creating freeform surfaces in HP PE/SolidDesigner: blending and lofting. The remainder of this article describes the basics of lofting.

Lofting means the (exact) interpolation of a set of points or curves by a smooth curve or surface. Fig. 3 shows examples of lofting. Lofting originated in ship design and was used a long time before computers were invented.

The mathematical solution of this problem leads to the definition of splines. There are many spline types, each having its specific advantages and disadvantages. The most common spline types are Bézier splines, B-splines, and NURBS.

For CAD applications the most general splines are NURBS, since they can represent analytics exactly. This can be important when it comes to intersections of splines and analytic surfaces. B-splines are NURBS with all weights equal to 1. They are more stable and faster in intersections but cannot represent analytics (except the plane) exactly. B-splines are made up of a sequence of Bézier pieces, connected according to their continuity at the transition points. We won't go into detail concerning spline mathematics here since there is abundant literature on this topic.[1,2]

In addition to the pure interpolation of points and curves, lofting allows the definition of tangent profiles at each 3D

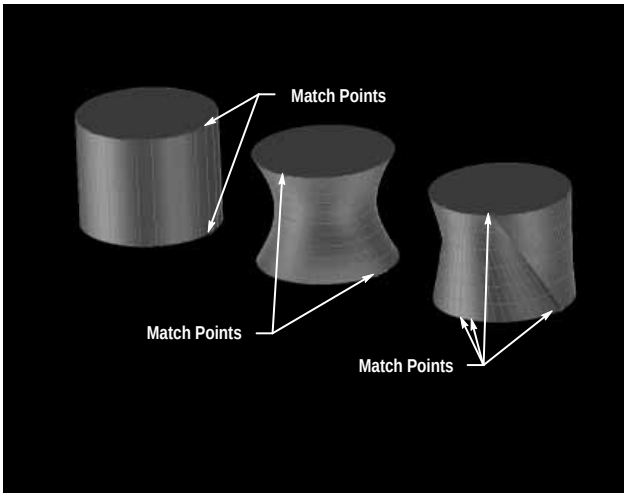


Fig. 6. Reparameterizing surfaces by specifying matching points on different input curves.

curve. A tangent profile is a kind of vector field along the given curve, as shown in Fig. 4. Both the directions and the lengths of the tangents influence the shape of the surface.

In practical applications the user normally wants to interpolate not only a series of single curves but also a series of multiply connected curves, as shown in Fig. 5. For this purpose HP PE/SolidDesigner connects the incoming profiles to a single B-spline curve. It is not required that a profile be smooth; it only needs to be C^0 continuous (closed). The C^0 locations in the profiles later correspond to edges in the complete model.

In addition to tangent profiles, the parameterization of the input curves is another important factor determining the shape of the lofted surface. In HP PE/SolidDesigner, parameterization can be influenced by splitting the input curves at arbitrary points (match points) and defining different length ratios in the subsequent profiles (Fig. 6). Within a curve segment, HP PE/SolidDesigner tries to create a parameterization according to the chord length of the curve (chordal parameterization).

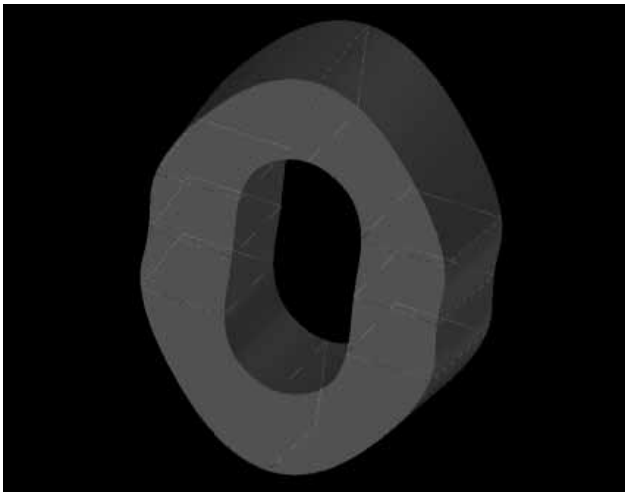


Fig. 7. Closed (periodic) surface created using lofting.

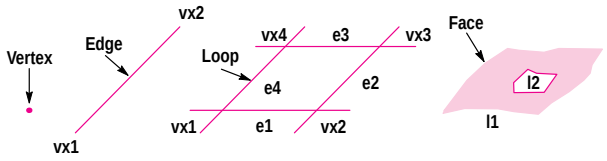


Fig. 8. Topological elements: vertex, edge, loop, face.

It is also possible to create closed (periodic) surfaces using lofting. In this case the first and last profiles are identical (Fig. 7).

Topology

Before explaining how topology is attached to the loft geometry, some definitions are needed (see Fig. 8):

- A vertex lies on a 3D point and can be viewed as the corner of a face.
- An edge is a bounded portion of a space curve. The bounds are given by two vertices.
- A loop represents a connected portion of the boundary of a face and consists of a sequence of edges.
- A face is a bounded portion of a geometric surface in space. The boundary is represented by one or more loops of edges.

Given a B-spline surface obtained from the spline library using the profile interpolation method, topology has to be built on this surface to get a loft body. As a boundary for the face, a loop consisting of four edges is created (Fig. 9). The edges lie on the first and last interpolation curves (e1 and e3) and on the left and right boundaries of the B-spline surface (e2 and e4).

The interpolation profiles don't have to consist of only one curve per profile. For more complex shapes different curves can be combined in a profile. It is necessary to generate a face for each matching set of curves. One way to do this is to use lofting to create a B-spline surface for each matching set and then build the appropriate faces on these surfaces. Because there is no exact specification of how the left and right boundaries of these B-spline surfaces should look there may be gaps between the faces (Fig. 10a). This would lead

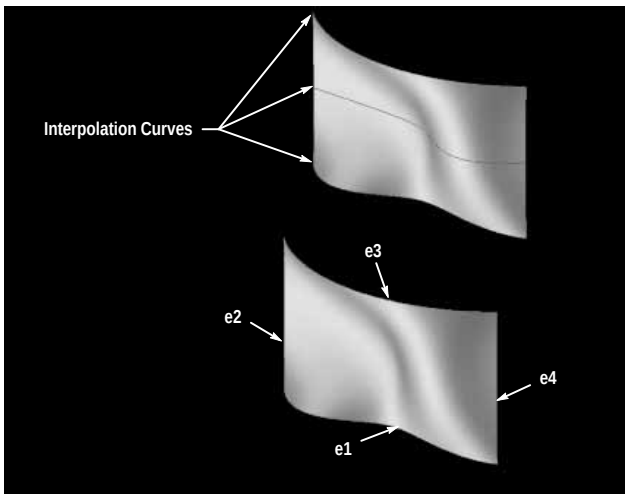


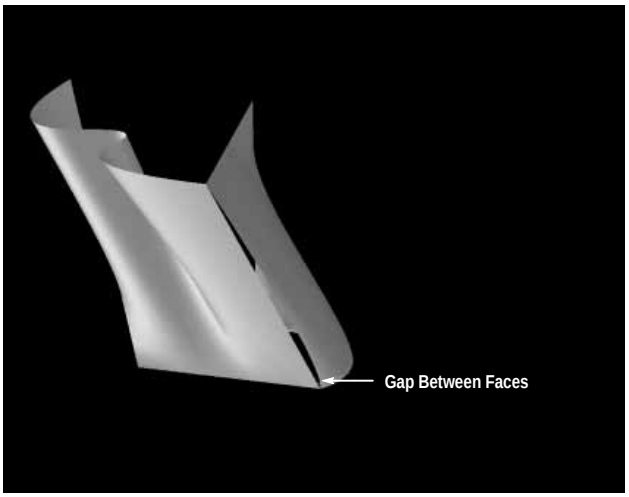
Fig. 9. Building topology on a B-spline surface by creating a boundary consisting of four edges.

to an illegal body, since all neighboring faces in a body have to share a common edge. When there are gaps between the faces no common edge can be found and it isn't possible to generate a valid body.

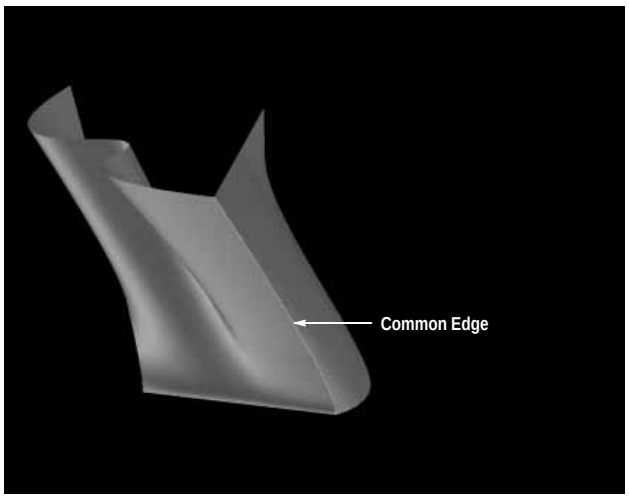
To eliminate gaps, the curves in one profile are joined temporarily and only one loft surface is generated. This B-spline surface then is split into appropriate parts at the start and end points of the interpolation curves. The faces are then built on the split surfaces. This ensures that there is no gap between the faces.

To match the correct curves or the correct portions of the curves it is necessary that all curves in a matching set have the same parameter interval. This is ensured by reparameterizing all curves belonging to the same matching set to the same (arbitrary) parameter interval. After this all curves of a profile are joined and the joined curves then automatically have the same parameter interval.

A valid solid body must describe a closed volume. For this reason only closed interpolation profiles are used. From these the lofting facility will generate faces forming a tube, which still has two open ends (Fig. 11). For each of the two ends a planar face is added. Theoretically these top and bottom faces can lie on any type of surface as long as the first and last interpolation profiles lie on the respective surfaces.



(a)



(b)

Fig. 10. (a) Illegal body with gaps between faces. (b) The system generates a common edge to eliminate gaps.

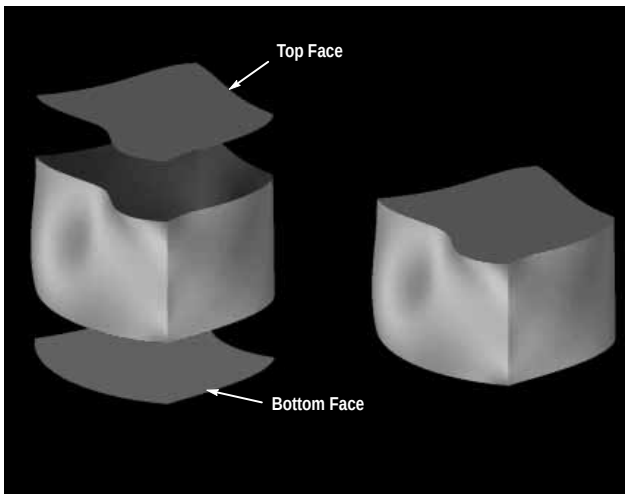


Fig. 11. Lofting generates a tube. Endfaces are added to make a solid body.

Lofting in HP PE/SolidDesigner

The spline library allows arbitrary 3D curves in space as interpolation profiles for lofting. To simplify the input process for the user, only planar profiles are allowed in the current release. These planar profiles can easily be generated in a workplane using 2D creation methods. All workplanes containing the profiles are gathered in a *workplane set*. The user specifies which set of curves should match in lofting. Different matching specifications will produce different loft results (Fig. 12).

Because the spline library only accepts B-spline curves as interpolation curves the analytic curves in the profiles have to be approximated by B-splines. Another reason for this is the above-mentioned joining of curves in a profile to obtain only one B-spline surface.

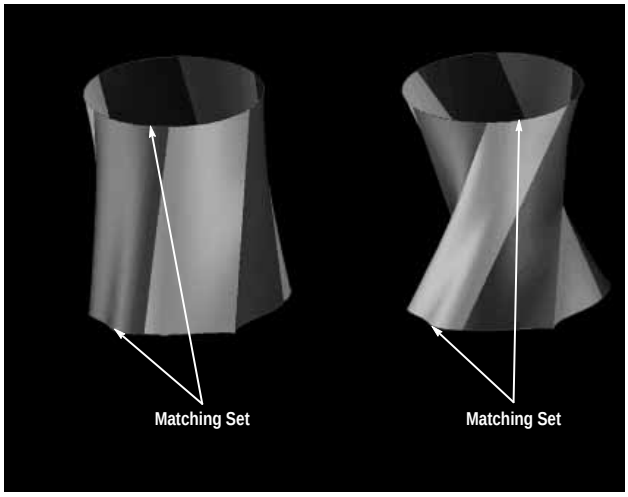


Fig. 12. Changing loft results by specifying different sets of matching curves.

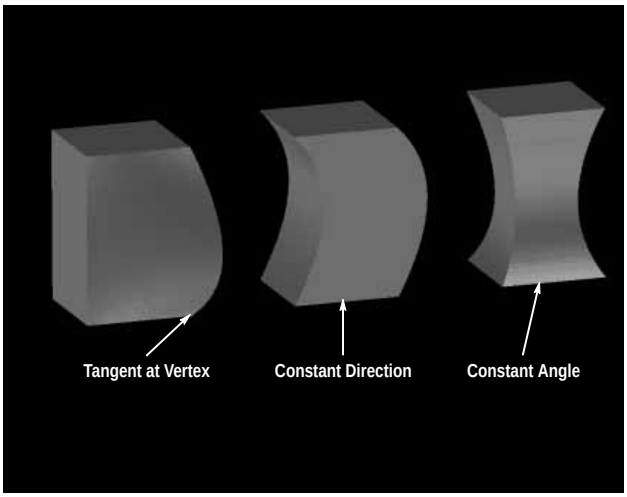


Fig. 13. Adding tangent conditions to change the shape of a body. (a) By defining a tangent direction to one or more vertices in the profile. (b) By specifying a constant direction for the entire profile. (c) By specifying an angle at one point of the profile. This angle is kept constant along the entire profile.

Another possibility for influencing the shape of the loft body is to add tangent conditions. In HP PE/SolidDesigner there are three different methods for doing this (Fig. 13):

- Define a tangent direction to one or more vertices in the profile.
- Specify a constant direction for the entire profile.
- Specify an angle at one point of the profile. This angle is kept constant along the entire profile.

For topology creation, especially face generation, the curves underneath the bounding edges of the faces have to be determined. Because the lofting algorithms only generate one B-spline surface, to get a properly connected tube this surface has to be split somehow. Because the single curves on the profiles have already been reparameterized to the same parameter interval for correct matching, this knowledge can be used to split the B-spline surface correctly. The boundaries of the split surface all lie on *isoparametric curves* of the loft surface. An isoparametric curve is a curve on a surface that has a constant u or v parameter value. In our case the loft direction is the v-parameter direction of the surface. This means that the left and right boundary curves of the faces are v-isoparametric curves. Splitting the surface along the

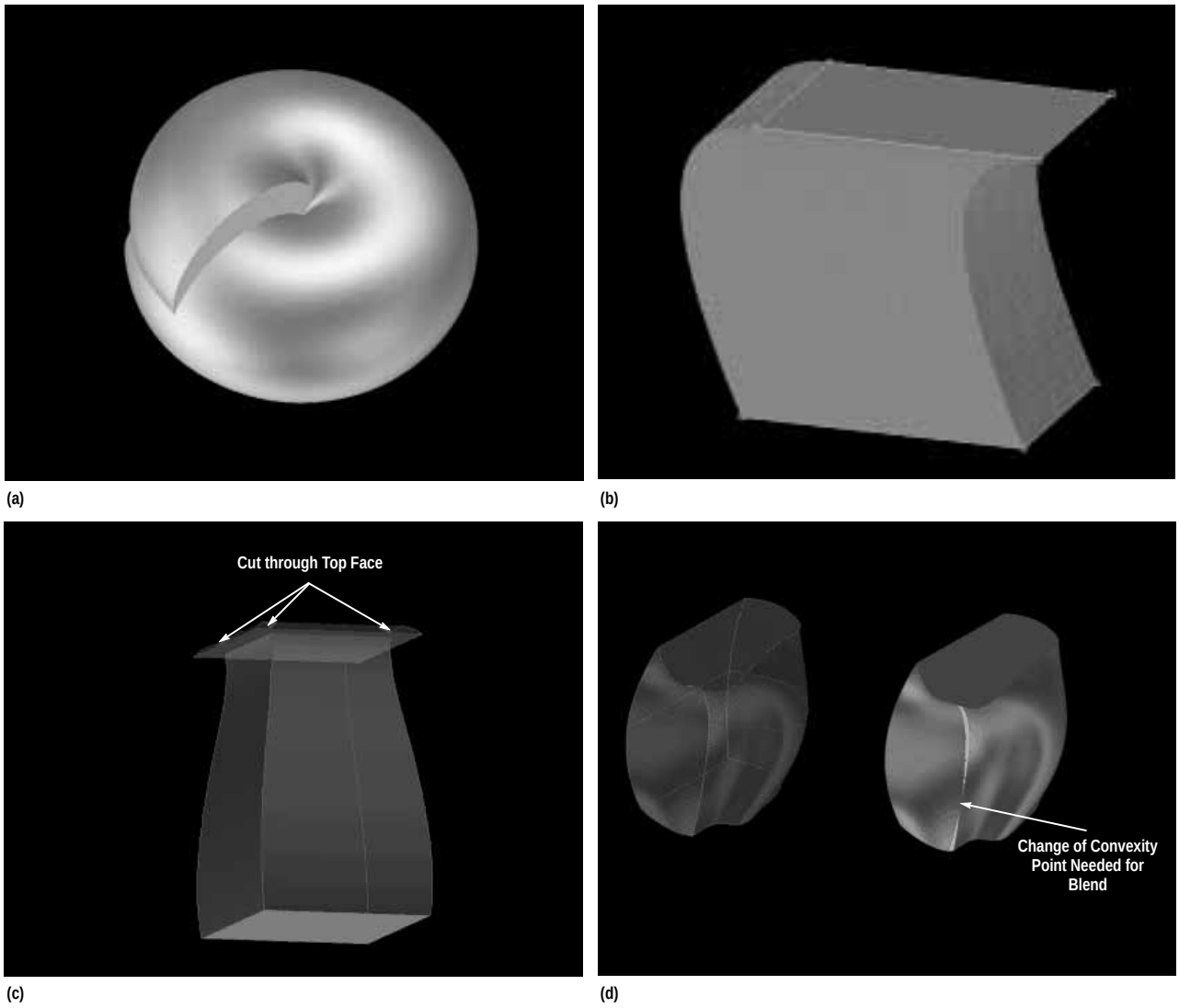


Fig. 14. HP PE/SolidDesigner checks for various properties that solid bodies shouldn't have. (a) Self-intersecting body. (b) Vanishing normals or derivatives. (c) Intersection with the top or bottom face. (d) Change of convexity at an edge.

v-parameter values of the start points and endpoints of the interpolation curves will result in the desired subsurfaces. The edges created on these v-isoparametric curves are always common to two neighboring faces.

Analytic Surface Type Detection

From a mathematical point of view the interpolation task that constitutes lofting is finished when the B-spline surface is created. From a CAD user's point of view the work is only partially finished. The reason is that it very often happens that a lofted body contains B-spline surfaces that represent analytical surfaces, mostly planes and cylinders. A CAD user wants to recognize these analytics in later processes for easier control in manufacturing. Data size, intersection performance, and stability are much better when dealing with analytics rather than approximated geometry. For these reasons a clever analytic detection algorithm is implemented in HP PE/SolidDesigner which replaces the B-spline strips by analytics after the B-spline creation and before the final topology is built.

The algorithm is based on the geometry of the input profiles. If curves of the same type are matched the basic definitions of these curves are compared (for example, the center of a circle, its radius, its starting point, etc.). Then, starting from the first two profiles, a corresponding analytic surface is built. In the next steps the other curves along the profiles (in the loft direction) are examined to see whether they fit this surface. If they do, the corresponding B-spline strip is exchanged and the neighboring topological information is adopted. This is done for each curve in the profile loop. Since the algorithm is based on the profiles and not on the lofted B-spline surface it is extremely fast and takes less than 1% of the time required for the lofting operation.

Special Cases

Lofting is a powerful tool for creating freeform surfaces in HP PE/SolidDesigner. On the other hand, there is a danger of creating surfaces that are not manufacturable or that have properties that can cause problems in later operations. For this reason, HP PE/SolidDesigner applies extra checks to ensure that the result of lofting is a clean body. These checks take extra time, normally more than the creation of the surface itself. HP PE/SolidDesigner therefore offers a button on the user interface to switch off these checks. It makes sense to switch the tests off in the surface design phase. For the final acceptance, however, it is recommended that the tests be run, since a corrupt model cannot be repaired later.

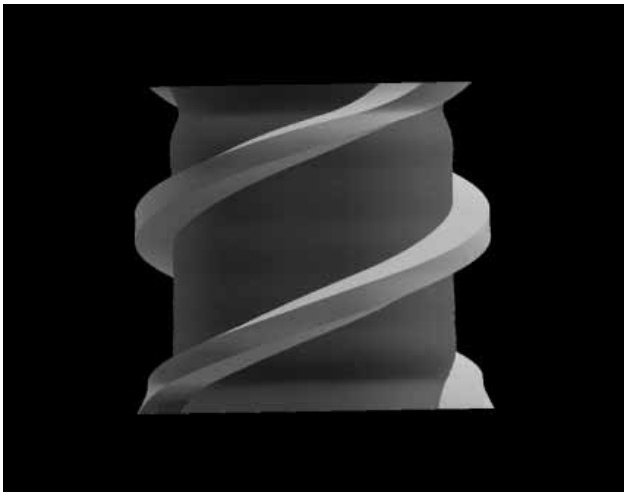


Fig. 15. Drill created using a special HP PE/SolidDesigner command to define a set of parallel workplanes, each turned by a given angle around an axis orthogonal to the base workplane.

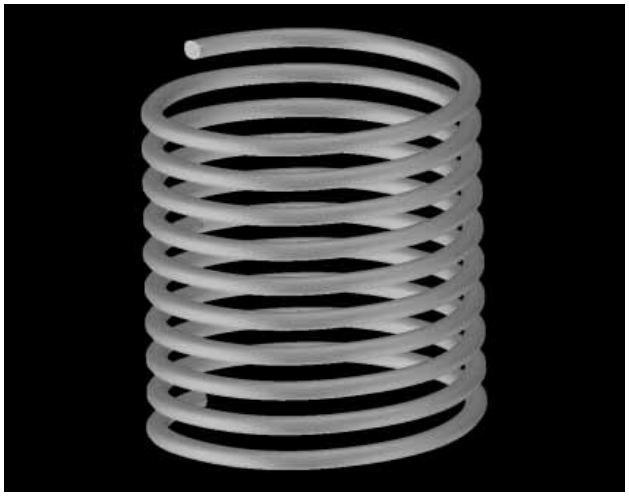


Fig. 16. Spring created using the workplane inclined command.

In the following examples we show the various properties a solid model shouldn't have. HP PE/SolidDesigner checks all of them and rejects the lofting operation if at least one of them appears. In the preview mode, the user can examine the object to find the root cause for the problem. The forbidden properties are:

- A self-intersecting body (Fig. 14a)
- Vanishing normals or derivatives (Fig. 14b)
- Intersection with the top or bottom face (Fig. 14c)
- Change of convexity at an edge (Fig. 14d). This test is always done and ensures that the specific edge can be blended later. HP PE/SolidDesigner will insert a topological vertex at the place where the convexity changes.

Practical Experience with Lofting

The most critical point in using lofting is the proper definition of the profiles and the workplanes. It turns out that in many real-life applications the profiles do not vary at all (e.g., helical constructions) or only a little. HP PE/SolidDesigner supports these surface classes by offering special

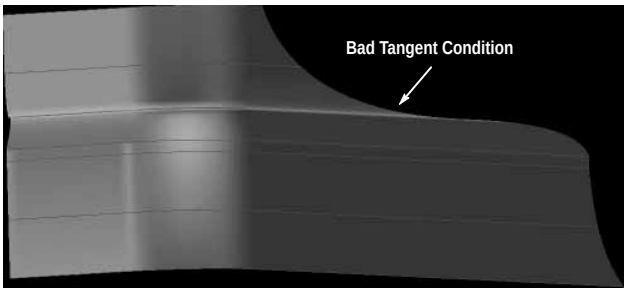
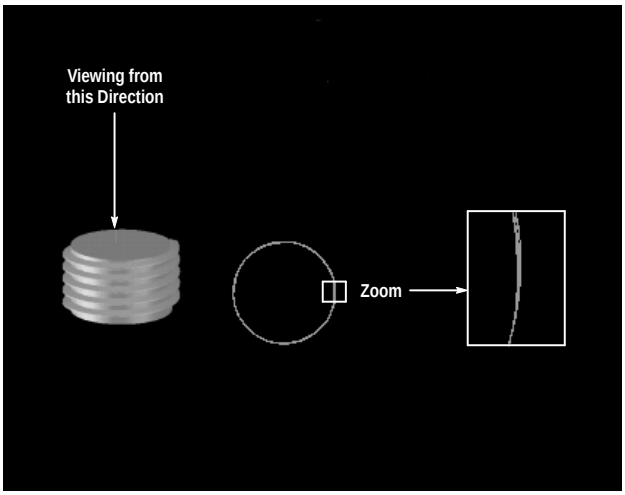


Fig. 17. Positioning too many profiles over too short a distance results in a wavy surface.

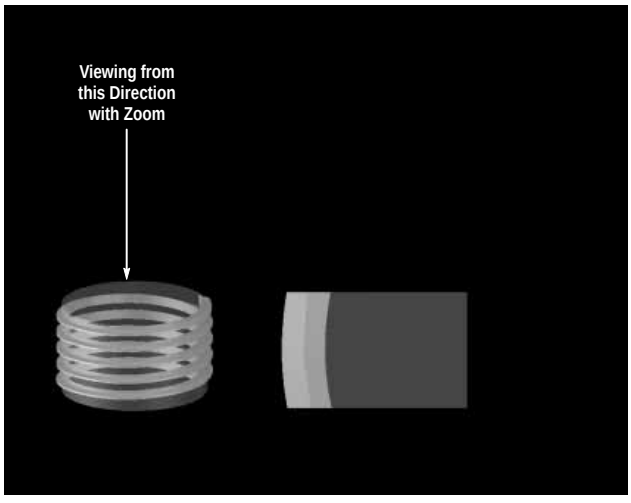
commands to create series of workplanes in the 3D space. These commands let the user define a set of parallel workplanes, each turned by a given angle around an axis orthogonal to the base workplane. Objects like drills can be created very easily (Fig. 15).

Using the "workplane inclined" command defines a set of workplanes at an angle to the base workplane. This is a way to create springs and other helical shapes (Fig. 16).

These special commands do not help in all situations. Sometimes the complete workplane set has to be defined by hand. Here it is important to know some basic behavior of the lofting algorithm to avoid subsequent problems with the Boolean topology engine. Often lofting is not used to create a completely new body but to cut off some existing geometry (loft remove) or to fill gaps (loft add). The most important property of lofting the user must keep in mind is that the surface starts oscillating if too many conditions (profiles, tangency conditions) are specified on too short a distance. The term "short" means short relative to the total object size. Positioning ten profiles over a distance of 100 millimeters causes no problems. Doing the same over a distance of one millimeter creates an awful surface. The same is true for the complexity of the profiles and the way the profiles change from one workplane set to another (Fig. 17).

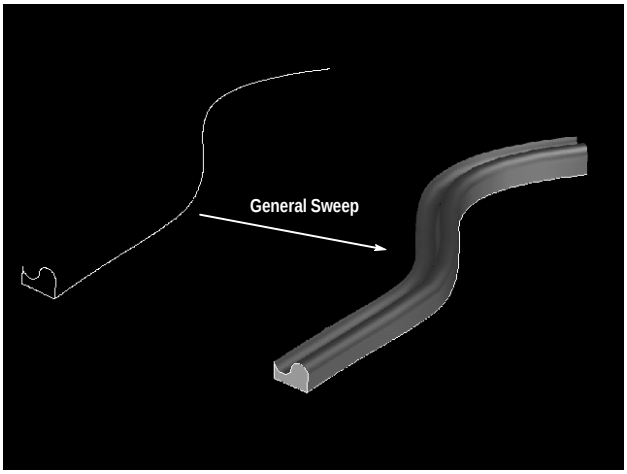


(a)

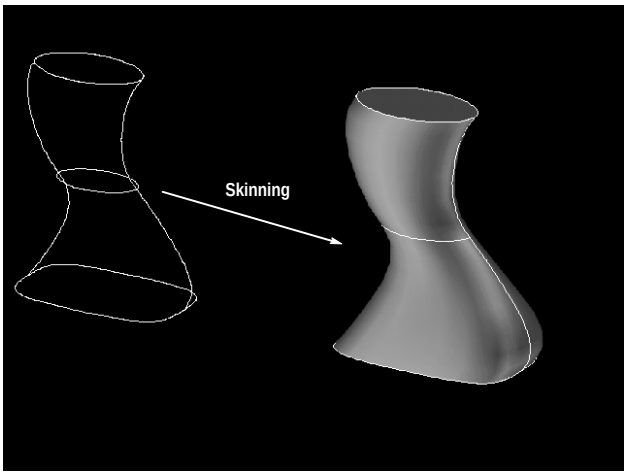


(b)

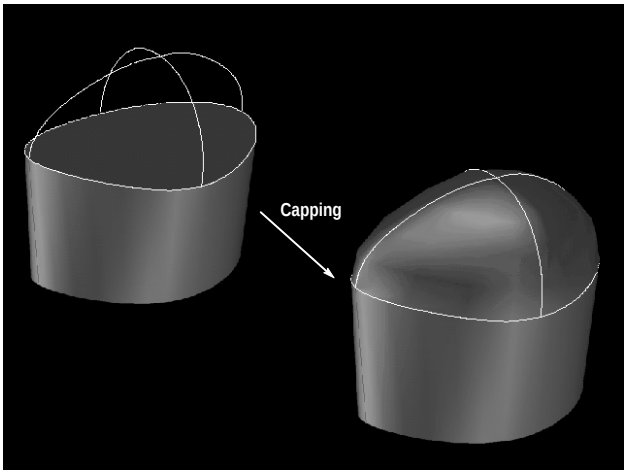
Fig. 18. (a) In creating a helical shape connected to a cylindrical shaft, if the helix base profile touches the cylinder a nonmanufacturable part results since the freeform helix oscillates around the cylinder surface. (b) If the helix base profile cuts into the cylinder a little the oscillating surface lies completely inside the cylinder and the unification of the two bodies will yield the expected result.



(a)



(b)



(c)

Fig. 19. (a) Sweeping. (b) Skinning. (c) Capping.

For this reason, one should never try to approximate other geometry using lofting in combination with a Boolean operation. It is much safer to create the loft tool body a little bigger to get clear intersections later. The example of Fig. 18 illustrates this. The task is to create a helical shape connected to a cylindrical shaft. The "workplane inclined" command is used to position the profiles for the loft. If the base profile touches the cylinder the unification of the lofted body and the cylinder will result in a nonmanufacturable part since the freeform helix oscillates around the cylinder surface (Fig. 18a).

However, if the profile cuts in a little the oscillating surface will lie completely inside the cylinder and the unification of both bodies will yield the expected result, as shown in Fig. 18b.

Summary

Lofting in HP PE/SolidDesigner is a powerful tool that enables the CAD user to create various freeform shapes within a solid model. The main task being solved by the user is the optimal selection of the profiles and clever positioning of the workplanes in the 3D space. With a little experience to gain familiarity with the behavior of the surface interpolation algorithms, many design tasks can be done in a short time. However, some tasks are cumbersome or nearly impossible using lofting, but are easily done using other HP PE/SolidDesigner capabilities. In electromechanical and mechanical engineering these tasks include mainly skinning, capping, and sweeping. Sweeping (Fig. 19a) is related to lofting since it means creating a surface by sweeping a profile along an arbitrary 3D curve. Skinning (Fig. 19b) is the task of defining a smooth surface through a net of 3D curves. Capping (Fig. 19c) means the replacement of a closed loop on a body by some smooth, tangentially connected surface; it is a subclass of skinning. Although these functionalities are the classical domain of surface modeling systems the open architecture of HP PE/SolidDesigner readily accommodates their implementation.

References

1. 1. G. Farin, *Curves and Surfaces for Computer-Aided Geometric Design*, Academic Press, 1988.

2. C. deBoor, *A Practical Guide to Splines*, Applied Mathematical Sciences no. 27, Springer, 1978.

Common Lisp as an Embedded Extension Language

A large part of HP PE/SolidDesigner's user interface is written in Common Lisp. Common Lisp is also used as a user-accessible extension language.

by Jens Kilian and Heinz-Peter Arndt

HP's PE/ME10 and PE/ME30 CAD systems contain an extension language based on the macro expansion paradigm. The user's input (commands and data) is separated into single tokens, each of which denotes a command, function, variable, macro name, number, string, operator, or other syntactic element. Commands, functions, and arithmetical expressions are evaluated by the language interpreter. Each macro name is associated with a macro definition, which is another token sequence (either predefined by the system or defined by the user). When the language interpreter encounters a macro name, it substitutes the corresponding token sequence (this process is called *expanding* the macro) and continues with the first token of the expansion.

Macro expansion languages are easy to implement and have been used in many applications where one would hardly expect to find an embedded language. For example, the TEX typesetting system contains a macro interpreter.

The HP PE/ME10 and PE/ME30 macro language includes powerful control constructs (such as IF/THEN/ELSE and LOOP/EXIT_IF/END_LOOP), local variables, and a mechanism for passing parameters to a macro when it is being expanded. These constructs make it possible to solve general programming problems. Because the HP PE/ME10 and PE/ME30 macro language is interpreted, programs can be developed in an interactive fashion and modifications can immediately be tried out. However, the resulting program is slower than a program written in a compiled language like C. HP PE/ME10 and PE/ME30 macros can be compiled to an intermediate form which executes faster than the pure interpreted version, but which is still slower than an equivalent C program.

One disadvantage of the HP PE/ME10 and PE/ME30 macro language is that it is nonstandard. No other application uses the same language, and programs written in it have to be ported when the user switches to another CAD system.

Common Lisp

Common Lisp was chosen as an extension language for HP PE/SolidDesigner because it is nonproprietary and widely used.

Surprising as it may be, Lisp is the second oldest high-level programming language still in common use. The only older one is FORTRAN. Lisp is to researchers in artificial intelligence what FORTRAN is to scientists and engineers.

Lisp was invented by John McCarthy in 1956 during the Dartmouth Summer Research Project on Artificial Intelligence. The first commonly used dialect was Lisp 1.5, but unlike FORTRAN (or any other imperative language) Lisp is so easy to modify and extend that over time it acquired countless different dialects. For a long time, most Lisp systems belonged to one of two major families, Interlisp and MacLisp, but still differed in details. In 1981, discussions about a common Lisp language were begun. The goal was to define a core language to be used as a base for future Lisp systems. In 1984, the release of Common Lisp: The Language[1] provided a first reference for the new language. An ANSI Technical Committee (X3J13) began to work on a formal standardization in 1985 and delivered a draft standard for Common Lisp in April 1992. This draft standard includes object-oriented programming features (the Common Lisp Object System, or CLOS). For a more detailed account on the evolution of Lisp, see McCarthy[2] and Steele and Gabriel.[3]

HCL, the implementation of Common Lisp used in HP PE/SolidDesigner, is derived from Austin Kyoto Common Lisp (itself descended from Kyoto Common Lisp). It corresponds to the version of the language described in reference 1, but already incorporates some of the extensions from reference 4 and the draft standard.

Applications of Extension Languages

Adding extension languages to large application programs has become a standard practice. It provides many advantages, some of which may be not as obvious as others. For the normal user of a system, an embedded programming language makes it possible to automate repetitive or tedious tasks. An inexperienced user can set it up as a simple record/playback mechanism, while "power users" can use it to create additional functionality. If the extension language has ties to the application's user interface, user-defined functionality can be integrated as if it were part of the original application.

If the application provides an API for adding extensions on a lower level, the extension language can itself be extended. This enables makers of value-added software to integrate their products seamlessly into the main application. As an example, the HP PE/SheetAdvisor application has been implemented within HP PE/ME30, offering a user interface consistent with the rest of the program.

As a final step, portions of the application can themselves be implemented in the embedded language. An example would be the popular GNU Emacs text editor, a large part of which is written in its embedded Lisp dialect.

A large part of HP PE/SolidDesigner, too, is written in its own extension language—about 30 percent at the time of writing. Most of this 30 percent is in HP PE/SolidDesigner's user interface.

Lisp in HP PE/SolidDesigner

Fig. 1 shows the major components of HP PE/SolidDesigner. The Lisp subsystem is at the very core, together with the Frame (operating system interface) and DSM (data structure manager, see article, page 51) modules. All other components including Frame and DSM are embedded into the Lisp subsystem. This indicates that each component provides an interface through which its operations can be accessed by Lisp programs.

The introduction of new functionality into HP PE/SolidDesigner is usually done in the following steps:

- Implement new data structures and operations in C++
- Add Lisp primitives (C++ functions callable from Lisp) for accessing the new operations
- Add action routines to implement new user-visible commands, using the Lisp interface to carry out the actual operations
- Add menus, dialog boxes, or other graphical user interface objects to access the new commands.

As long as the Lisp interface—the primitive functions—is agreed to in advance, this process can be parallelized. A user interface specialist can work on the action routines and menus, calling dummy versions of the interface functions.

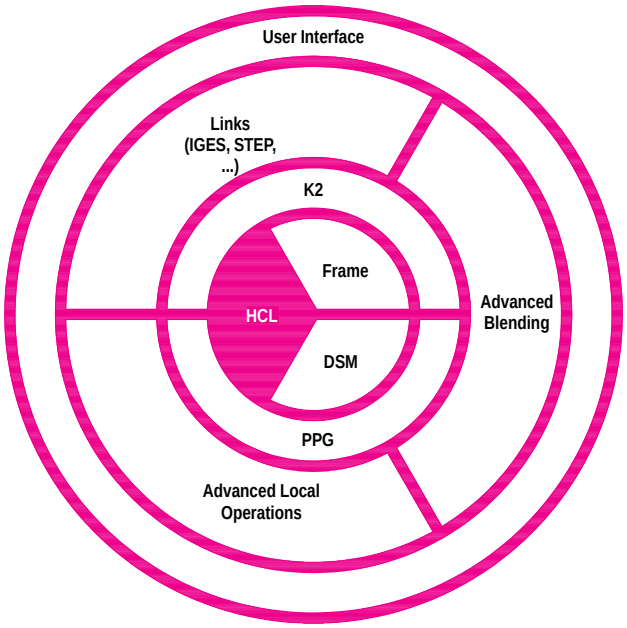


Fig. 1. HP PE/SolidDesigner system architecture. HCL is the Common Lisp subsystem. All components including Frame (operating system interface) and DSM (data structure manager) have interfaces to Lisp. K2 is the solid modeling kernel. PPG is the planar profile generator.

The article on page 14 describes, from a user interface developer's perspective, how action routines are written and how menus and dialogs are created. The mechanisms used there are not part of the Common Lisp standard but are extensions provided by the HCL dialect.

Action Routines

Action routines implement the commands that a user types or issues via user interface elements to HP PE/SolidDesigner. Commands are identified by their names, which are Lisp symbols evaluated in a special manner (similar to the SYMBOL-MACROLET facility in the Common Lisp Object System). Each action routine is actually an interpreter for a small language, similar in syntax to the command language used in HP PE/ME10 and PE/ME30. Like HP PE/ME10 and PE/ME30 commands, action routines can be described by their syntax diagrams. Fig. 2 contains the syntax diagram for a simplified version of HP PE/SolidDesigner's exit command. Below the syntax diagram is a state transition graph which shows how the command will be processed.

The definition of an action routine corresponds closely to its syntax diagram. The defining Lisp expression, when evaluated, generates a normal Lisp function that will traverse the transition graph of the state machine when the action routine is run. For example, the following is an action routine corresponding to the syntax diagram of Fig. 2:

```
(defaction simple_exit

 (flag)      ; local variable

 (; state descriptions

  (start nil
       "Terminate PE/SolidDesigner?"
       nil

       (:yes     (setq flag t)   answer-yes end)
       (:no      (setq flag nil) answer-no  end)
       (otherwise (display_error "Enter either :YES or :NO.")  nil start))

  (end (do-it)
     nil
     nil))

 (; local functions

  (do-it ()
       (when flag
        (quit)))))
```

As can be seen in this example, an action routine can have local variables and functions. Local variables serve to carry information from state to state. Local functions can reduce the amount of code present in the state descriptions, enhancing readability.

When HCL translates this action routine definition, it produces a Lisp function which, when run, traverses the state transition graph shown in Fig. 2b. If a state description contains a prompt string, as in the start state in the example, the translator automatically adds code for issuing the prompt and reading user input. Effectively, the translator converts the simple syntax diagram into the more detailed form.

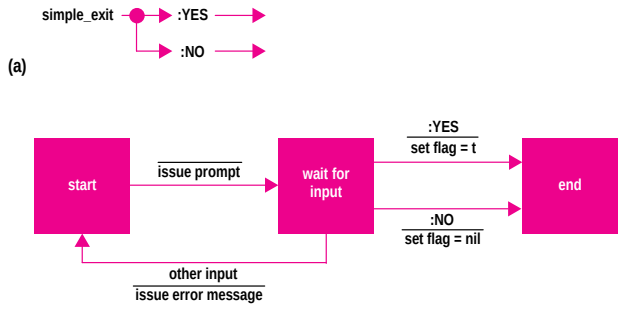


Fig. 2. (a) Simplified syntax of the exit command. (b) State transition diagram for the exit command.

For the example action routine, the translator produces a Lisp function definition much like the following:

```
;; Declarations of some external functions, for more efficient calling

(proclaim '(function get-parameter  (t t) t))
(proclaim '(function match-otherwise (t) t))
(proclaim '(function trigger-action-state-transition-event (t &optional t)
   t))

;; Transformed action routine

(defun simple_exit (&rest argument-list &aux input)

 (let (flag)              ;; local variable

   (labels ((do-it ()      ;; local function
              (when flag
                (quit))))

         (block nil
           (tagbody

            ;; label for state "start"
            1

            ;; prompting in state "start"
            (setq input (get-parameter argument-list "Terminate HP PE/
            SolidDesigner?"))

            ;; pattern matching in state "start"
            (cond ((equal input :yes)

                   (setq flag t)   ;; action taken
                   (trigger-action-state-transition-event  'answer-yes)

                   (go 0))        ;; transition to "end" state

                  ((equal input :no)

                   (setq flag nil) ;; action taken
                   (trigger-action-state-transition-event  'answer-no)

                   (go 0))        ;; transition to "end" state

                  ((match-otherwise input)

                   (display_error "Enter either :YES or :NO.")

                   (go 1)))       ;; transition to "start" state

            ;; label for state "end"
            0

           ;; initial action for state "end"
           (do-it)

           ;; exit from action routine
           (return))))))
```

Transitions in the state machine are transformed into goto statements within the function's body. The conditional construct cond represents decisions, like the three-way branch in state start. Before each state transition, the code can trigger an external event to enable graphical feedback in menus or dialogs.

The actual translation is somewhat more complicated because errors and other exceptional events must be taken into account. The translator also adds code to support debugging and profiling of an action routine. This code is stripped out when building a production version of HP PE/SolidDesigner.

Compiling Lisp Programs

It has often been said that Lisp is inherently slow and cannot be applied to application programming (one common joke is that the language's name is an acronym for "Large and Incredibly Slow Programs"). This is not true. Even very early versions of Lisp had compilers.[3] Lisp systems have even beaten FORTRAN running on the same machine in terms of numerical performance.

In HCL, the Lisp compiler takes a Common Lisp program and translates it into an intermediate C++ program, which is then compiled by the same C++ compiler that is used to translate the nonLisp components of HP PE/SolidDesigner. This approach has several advantages:

- The Lisp compiler can be kept small and simple (only 12,500 noncomment lines of code, less than 5% of the total amount of Lisp code)
- The Lisp compiler does not need to be retargeted when porting to a different machine architecture
- The Lisp compiler does not need to fully optimize the generated code; this task can be left to the C++ compiler
- The generated code is fully call and link compatible with the rest of the system
- The generated code can be converted to a shared library and dynamically loaded into a running HP PE/SolidDesigner.

The Lisp compiler is itself written in Lisp. Bootstrapping a new compiler version is easy because an interpreter is available.

The calling conventions for compiled Lisp functions are such that interpreted and compiled functions can transparently call each other. This allows keeping most of the Lisp code in compiled form, even when using the interpreter to develop new programs.

Continuing the above example, here is the C++ code that the Lisp compiler produces for the simplified translated action routine (reformatted for better readability):

```
// Header file declaring standard Lisp data structures and functions
// (for example, LOBJP is the type of a generic pointer-to-Lisp-object)

#include <cmpinclude.h>

// Declarations for the compiled code (normally written to a separate file,
// included here for clarity)

static void L1(...);                    // Functions defined in this file
static void L2(LOBJP*);
```

```
static char *Cstart;                    // Data for communication with the Lisp
                                        // loader

static int Csize;
static LOBJP Cdata;
static LOBJP VV[14];                    // Run-time Lisp objects

static void LnkT13() ;                  // Links to external Lisp functions
static void (*Lnk13)() = LnkT13;        // (see below for an explanation)
static void LnkT11() ;
static void (*Lnk11)() = LnkT11;
static LOBJP LnkTLI10(LOBJP ) ;
static LOBJP (*LnkLI10)(LOBJP ) = LnkTLI10;
static LOBJP LnkTLI9(int narg, ...) ;
static LOBJP (*LnkLI9)(int narg, ...) = LnkTLI9;
static LOBJP LnkTLI8(LOBJP , LOBJP ) ;
static LOBJP (*LnkLI8)(LOBJP , LOBJP ) = LnkTLI8;

// Initialization function, called immediately after the file is loaded

void example_initialize(char *start, int size, LOBJP data)
{
  // Reserve space on the Lisp stack

  register LOBJP* base=vs_top;
  register LOBJP* sup=base+0;
  vs_top=sup;
  vs_check;

  // Store data supplied by the loader, including Lisp objects
  // that were extracted from the original source code and that
  // will be needed at run-time (e.g., strings and symbols).

  Cstart=start;
  Csize=size;
  Cdata=data;
  set_VV_data(VV,14,data,start,size);

  // Link the compiled function "L1" to the Lisp symbol stored in VV[6],
  // which is "SIMPLE_EXIT".

 MFnew(VV[6],(void(*)())L1,data);//

  // Restore Lisp stack

  vs_top=vs_base_mod=base;
}

// Compiled function SIMPLE_EXIT

static void L1(...)
{
  register LOBJP*base=vs_base;      // Reserve space on the Lisp stack
  register LOBJP*sup=base+3;
  vs_check;

  { LOBJP V1;                       // Fetch ARGUMENT-LIST from the Lisp
                                    // stack
    vs_top[0]=Cnil;
    { LOBJP *p=vs_top;
      for(;p>vs_base;p--)p[-1]=MMcons(p[-1],p[0]);
    }
    V1=(base[0]);
    vs_top=sup;
    { LOBJP V2;                     // Set up variables INPUT and FLAG
      V2= Cnil;
      base[1]= Cnil;
    T3::;                           // Label "1" in TAGBODY
      V2= (*(LnkLI8))((V1),VV[0]);  // (GET-PARAMETER ARGUMENT-LIST "...")
      if(!(equal((V2),VV[1]))){     // First clause of COND construct
        goto T8;
      }
      base[1]= Ct                   // (SETQ FLAG T)
      (void)((*(LnkLI9))(1,VV[2])); // (TRIGGER-...-EVENT 'ANSWER-YES)
      goto T4;                      // (GO 0)
    T8::;                           // Second clause of COND construct
      if(!(equal((V2),VV[3]))){
        goto T14;
      }
      base[1]= Cnil;                // (SETQ FLAG NIL)
      (void)((*(LnkLI9))(1,VV[4])); // (TRIGGER-...-EVENT 'ANSWER-NO)
      goto T4;                      // (GO 0)
    T14::;                          // Third clause of COND construct
      if(((*(LnkLI10))((V2)))==Cnil){
        goto T4;
      }
      base[2]= VV[5];               // (DISPLAY-ERROR "...")
      vs_top=(vs_base=base+2)+1;
      (void) (*Lnk11)();
      vs_top=sup;
      goto T3;                      // (GO 1)
    T4::;                           // Label "0" in TAGBODY
      vs_base=vs_top;               // Call (DO-IT), passing a pointer to
      L2(base);                     // the lexical variables of SIMPLE_EXIT
      vs_top=sup;
      base[2]= Cnil;                // Return from SIMPLE_EXIT
      vs_top=(vs_base=base+2)+1;
      return;
    }
  }
}

// Compiled local function DO-IT

static void L2(LOBJP*base0)
{
  register LOBJP*base=vs_base;      // Reserve space on the Lisp stack
  register LOBJP*sup=base+1;
  vs_check;
  vs_top=sup;
  if((base0[1])==Cnil){             // Condition: lexical variable FLAG
    goto T26;
  }
  vs_base=vs_top;                   // (QUIT)
  (void) (*Lnk13)();
  return;
 T26::;
  base[0]= Cnil;                    // Return from DO-IT
  vs_top=(vs_base=base+0)+1;
  return;
}

// Links to external functions. These functions are called indirectly, via
// C++ function pointers.  At the first call, the corresponding compiled
// function is looked up and stored in the function pointer, thus avoid-
// ing the Lisp calling overhead on subsequent calls.

static void LnkT13 ()
{ // QUIT; called via normal Lisp calling conventions

  call_or_link(VV[13],(int *)&Lnk13);
}
```

```
static void LnkT11()
{ // DISPLAY-ERROR; called via normal Lisp calling conventions

  call_or_link(VV[11],(int *)&Lnk11);
}

static LOBJP LnkTLI10(LOBJP arg0)
{ // MATCH-OTHERWISE; declared to take exactly one parameter, which
  // can be passed without using the Lisp stack.

  return(LOBJP)call_fproc(VV[10],(int*)&LnkLI10,1,arg0);
}

static LOBJP LnkTLI9(int narg, ...)
{ // TRIGGER-ACTION-STATE-TRANSITION-EVENT; declared to take one
  // fixed and one optional parameter, which can be passed without using
  // the Lisp stack.

  va_list ap;
  va_start(ap, narg);
  LOBJP result=(LOBJP)call_vproc(VV[9],(int*)&LnkLI9,narg,ap);
  va_end(ap);
  return result;
}

static LOBJP LnkTLI8(LOBJP arg0, LOBJP arg1)
{   // GET-PARAMETER; declared to take exactly two parameters, which
    // can be passed without using the Lisp stack.

  return(LOBJP)call_fproc(VV[8],(int*)&LnkLI8,2,arg0,arg1);
}
```

This example illustrates several important properties of compiled Lisp code. First, the C++ code still has to access Lisp data present in the original program; for example, it has to attach a compiled function to a Lisp symbol naming that function. Second, parameter passing for Lisp functions is usually done via a separate stack, but the overhead for this can be avoided by declaring external functions. In a similar way (not shown here), the overhead of using Lisp data structures for arithmetic can be avoided by introducing type declarations (which are not compulsory as in C++). Third, some Lisp constructs (e.g., lexical nesting of function definitions) have no direct C++ equivalent.

Compiling a Lisp program can have quite a dramatic impact on its performance. HP PE/SolidDesigner takes about one half to two minutes to start on an HP 9000 Series 700 workstation. If all the Lisp files are loaded in uncompiled form, start time increases to between one half and one hour.

Conclusion

A large part of HP PE/SolidDesigner is written in Common Lisp. To the developers, this approach offered a very flexible, interactive mode of programming. The finished programs can be compiled to eliminate the speed penalty for end users.

Common Lisp is also used as a user-accessible extension language for HP PE/SolidDesigner. It is a standardized, open programming language, not a proprietary one as in HP PE/ME10 and PE/ME30, and the developers of HP PE/SolidDesigner believe that this will prove to be an immense advantage.

References

1. G.L. Steele, Jr., S.E. Fahlman, R.P. Gabriel, D.A. Moon, and D.L. Weinreb, *Common Lisp: The Language*, Digital Press, 1984.

2. J. McCarthy, "History of LISP," in R.L. Wexelblat, ed., *History of Programming Languages*, ACM Monograph Series, Academic Press, 1981. (Final published version of the *Proceedings of the ACM SIGPLAN History of Programming Languages Conference*, Los Angeles, California, June 1978.)

3. G.L. Steele, Jr. and R.P. Gabriel, "The Evolution of Lisp," *Proceedings of the Second ACM SIGPLAN History of Programming Languages Conference*, Cambridge, Massachusetts, April 1993. pp. 231-270.

4. G.L. Steele, Jr., S.E. Fahlman, R.P. Gabriel, D.A. Moon, D.L. Weinreb, D.G. Bobrow, L.G. DeMichiel, S.E. Keene, G.Kiczales, C. Perdue, K.M. Pitman, R.C. Waters, and J.L. White, *Common Lisp: The Language, Second Edition*, Digital Press, 1990.

Boolean Set Operations with Solid Models

The Boolean engine of HP PE/SolidDesigner applies standard and nonstandard Boolean set operations to solid models to perform an impressive variety of machining operations. Parallel calculation boosts performance, especially with multiprocessor hardware.

by Peter H. Ernst

Machining operations like punch, bore, and others play an important role in the function set of contemporary CAD systems. In HP PE/SolidDesigner, the impressive variety of machining commands are driven by a single topology engine, often referred to as *the Boolean engine*.

It might seem that the algorithm used by the Boolean engine would be extremely complex and esoteric, and this is indeed true in some respects. The underlying principles, however, are simple.[1] Most of this article demonstrates this by taking a fairly intuitive look at the internal machinery. This will provide a road map for the second, more technical part of the article, in which some key algorithms are explained in greater depth. Finally, some unusual applications of the Boolean engine are briefly mentioned.

Different Flavors of Solids

Before exploring the internals of the Boolean engine, let's take a look at the objects that it works on. These objects are called solids, or simply bodies. Solids, in our terms, are mathematical boundary representation (B-Rep) models of geometric objects. Fig. 1 shows a B-Rep model of a cylinder.

Usually several categories of solids are distinguished based on their manifold characteristics. For our purposes we just need to know that *manifold* solids represent real objects and *nonmanifold* solids are impossible in some way. Manifold bodies are of general interest, since they can be manufactured. Fig. 2 shows a screwdriver representing the class of manufacturable bodies.

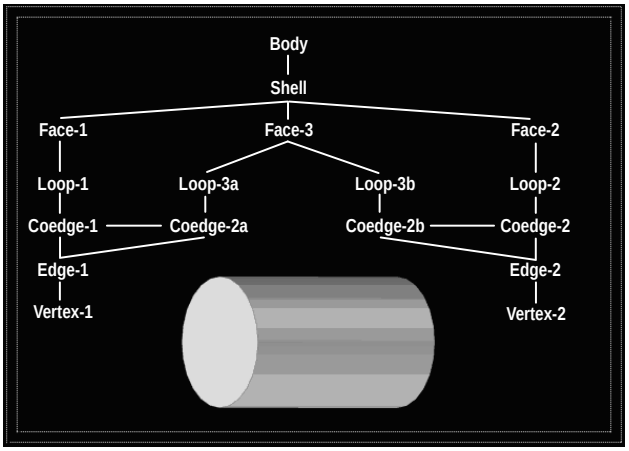


Fig. 1. A boundary representation (B-Rep) model of a cylinder.

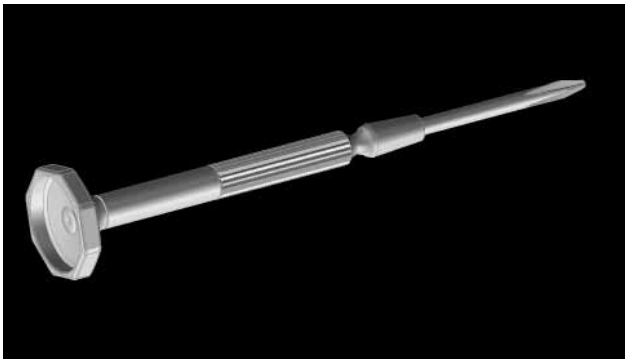


Fig. 2. A screwdriver representing the class of manufacturable bodies.

The class of nonmanifold bodies is the realm of the impossible bodies. These bodies cannot be manufactured because the material thickness goes to zero (that the thickness goes to zero is a consequence, not a cause of the nonmanifoldness). Nevertheless, they have have some importance as conceptual abstractions or simplifications of real (manifold) solids. Nonmanifold solids sometimes are (conveniently) generated as an intermediate step in the design process. They are also important to various simulation applications, and sometimes to finite-element analysis and NC machining. Fig. 3 shows a selection of nonmanifold bodies. To the left is

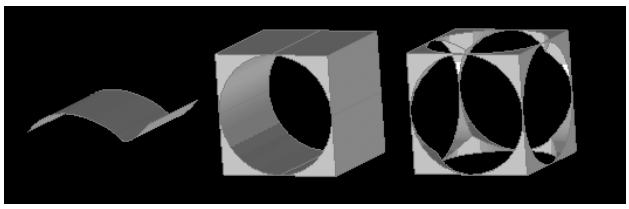


Fig. 3. Bodies that are nonmanufacturable because of (left) zero thickness in general, (center) zero thickness at edges, and (right) zero thickness at vertices.

a *sheet*, which has zero thickness in general. The middle solid is *edge nonmanifold*, having zero thickness at edges, and the right solid is *vertex nonmanifold*, having zero thickness at vertices.

The Boolean engine in its different guises is used to change bodies by the rules of Boolean set operations. In other words, it is able to combine two volumes using one of the three standard operators: subtract, unite, or intersect. The operation is performed on solids in the same way as on the sets of mathematical set theory. The effects of the standard operations on sets and volumes are illustrated in Fig. 4. The two bodies at the top of the picture are combined in three ways, using the three standard Boolean operations. The result of each Boolean operation is shown at the bottom.

An Intuitive Approach to the Boolean Engine

Now that we are equipped with the right background, we can explore the various stages of the Boolean algorithms. To do this we will use a thought experiment (such experiments are widely acknowledged as safe and cheap). To perform this experiment we only need some paint, a sharp knife, and some imagination.

Coloring. In the first stage both solids participating in the Boolean operation are filled with different colors, let's say yellow for one and blue for the other. Fig. 5 shows two bodies that have been set up for a Boolean operation and colored according to our rule. Let's assume that, unlike real solids, they can permeate each other without problems. Since the Boolean operation hasn't been performed yet the picture still shows two disjoint solids that just happen to overlap. To show what's going on inside the bodies, the yellow body has been made transparent.

Now we mark the lines where the two bodies permeate each other, let's say with red color. The red lines in Fig. 6 are called the *intersection graph*. The two solids are still disjoint.

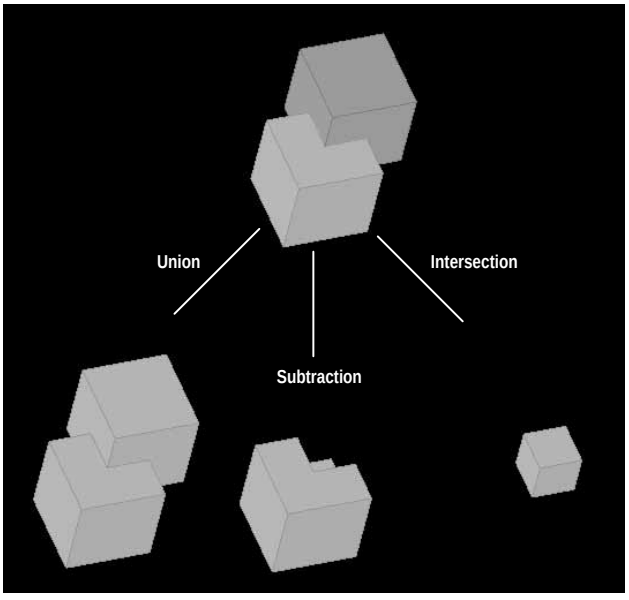


Fig. 4. Results of applying the standard Boolean set operations to two solid bodies.

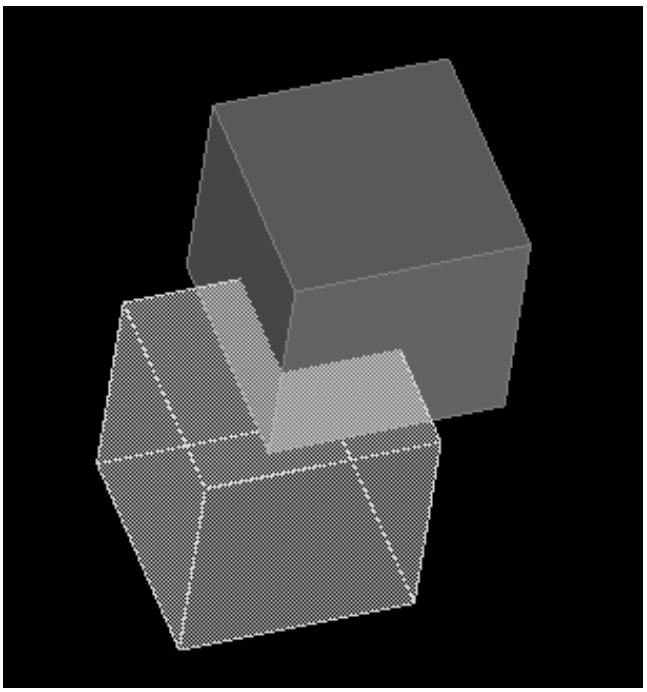


Fig. 5. Two disjoint solids that happen to overlap.

Making Soap Bubbles—Cellular Bodies. In the second stage we *knit* both solids together using the intersection graph. A structure very similar to those formed by soap bubbles is created, as shown in Fig. 7. The two solids now hang together at the intersection graph. In the space where both bodies overlap a green color can be seen. This is the mixture of yellow and blue. To get a better vision of the geometric situation some faces have been made transparent.

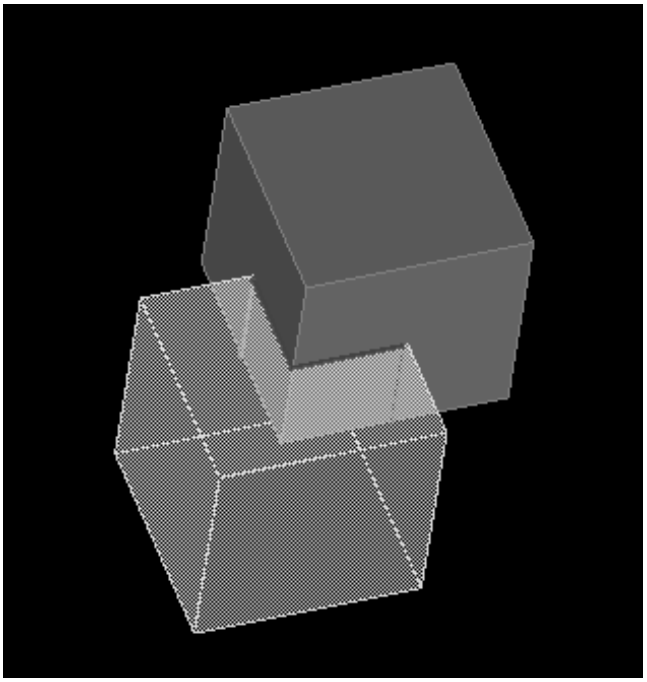


Fig. 6. Intersection graph (red).

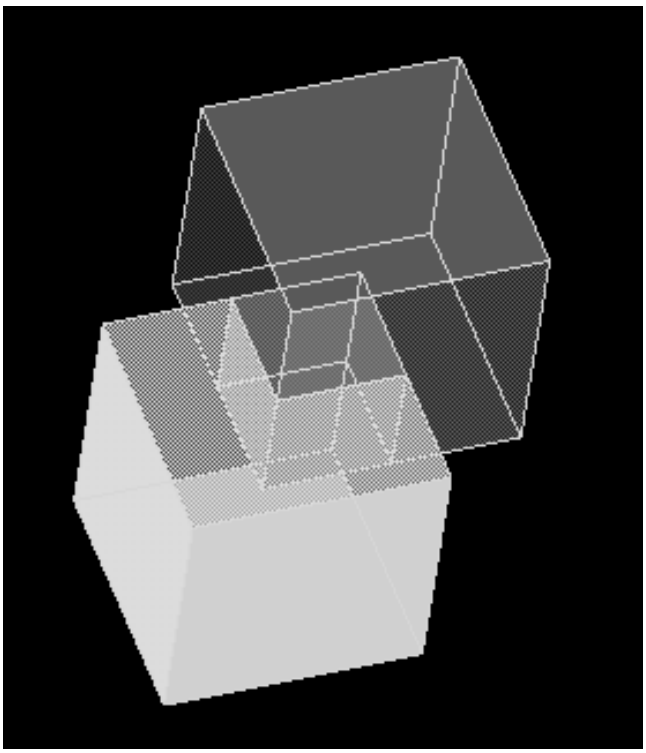


Fig. 7. Result of knitting the two bodies together at the intersection graph. Choosing a Boolean operation is now equivalent to deciding which colors to keep and which to delete.

Getting Rid of the Wrong Colors. In the third and last stage of our imaginary process not much is left to do. Up to now we have not said which kind of Boolean operation (union, subtraction, or intersection) we wanted. Now is the time to decide.

To get the desired result we simply pick the appropriate color and get rid of all volumes of a different color than the one we picked. Initially we chose two colors—blue and yellow—so we will find three colors in our soap bubble cluster: blue, yellow, and green. In regions where blue and yellow volumes overlap we get green. The table below shows which colors will be kept or deleted from the body depending on the particular type of Boolean operation we choose.

	Keep	Delete
Union	blue and yellow	green
Subtraction	yellow	green and blue
Intersection	green	blue and yellow

Easy, isn't it? Pat yourself on the back (and clean up the mess of paint and chipped-off pieces).

Technical Talk: The Boolean Algorithm

In the preceding example we only had to mark the lines where the color changes to obtain the intersection graph. The Boolean engine algorithm that does this is a bit more complex. To understand it we must again look at the mathematical representation of a solid. In Fig. 1 we have seen the general data structure layout of a cylinder. That sketch, however, lacks any explicit references to geometry. In HP PE/SolidDesigner's B-Rep structure, three base classes of geometries are used: points, curves, and surfaces. The last two have several subclasses. For example, a curve can be a straight line, circle, ellipse, or spline. In the following discussion the geometric subclasses are used for illustration purposes, but the Boolean algorithm itself does not depend on any specific geometry types, since it is implemented in a generic way.

Each geometry class has a corresponding topological carrier that puts it into perspective in the context of a solid model. The table below shows this relationship:

Topology		Geometry
Vertex	←	Point
Edge	←	Curve
Face	←	Surface

The topological entities face and edge are *smart* carriers because they not only *hold* their geometries, but also *bound* or *trim* them. To understand what this means we must realize that most geometries are of infinite extent, and even if they are finite only a small segment might be of interest.

Fig. 8 exemplifies the relationship between topology and geometry. Looking at the cylinder (sf3), notice that only a segment of the otherwise infinite cylindrical surface is used. This segment is called a face (fa3). Likewise, only two circular regions of the otherwise infinite planes sf1 and sf3 are used to close the cylinder. The circular regions are face fa1 and face fa2. (Note: The top and bottom faces of the cylinder have been lifted off a bit for better demonstration. The double yellow edges coincide in reality.)

The concept of trimmed surfaces is essential for the next section, because it introduces some unexpected complications when constructing the intersection graph.

Constructing the Intersection Graph. Earlier we simply used an excellent pattern recognizer called the human brain to find the lines where the color changes. Teaching this ability to a computer involves a considerable amount of mathematics.

Fig. 9 shows the construction of one segment of an intersection graph (a graph edge). The drawing shows two intersecting surfaces sf1 and sf2 carrying two faces fa1 and fa2. To construct the graph edge (the piece of the intersection track inside both faces) the following steps are required:

- The two unbounded surfaces sf1 and sf2 are intersected, giving the intersection track (track).

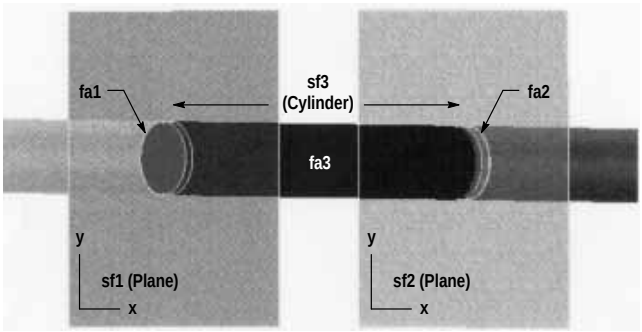


Fig. 8. An example of the relationship between topology and geometry. Faces and edges bound or trim their geometries, which consist of infinite curves and surfaces.

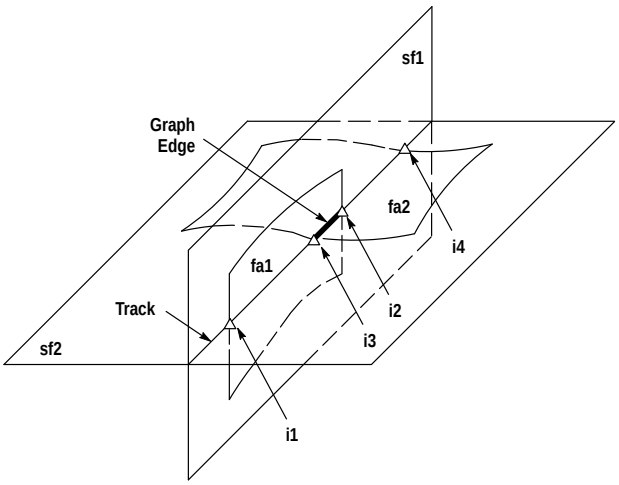


Fig. 9. Construction of one segment of an intersection graph (a graph edge).

- The edges of fa1 are intersected with surface sf2 to yield the edge/surface intersection points i1 and i2. Similarly, the edges of fa2 are intersected with sf1 giving the intersection points i3 and i4.
- The intersection points are ordered along the track.
- The ordered points are examined for their *approach values*. The approach values simply tell if a face is entered or left when passing a particular point. This information can be used to deduce the containment of a segment of the intersection graph with respect to its generating faces. The approach and containment values for the intersection points in the previous drawing are:

Point	Containment with respect to:		
	Approach	fa1	fa2
		outside	outside
i1	entering fa1		
		inside	outside
i2	entering fa2		
		inside	inside
i3	leaving fa1		
		outside	inside
i4	leaving fa2		
		outside	outside

- The segments of the intersection graph inside both faces are used to create the graph edge(s) of a particular intersection. In this example only the segment bounded by i2 and i3 fulfills this condition.

Parallelism. The complete intersection graph of two bodies is obtained by pairwise intersection of faces selected from both solids. The number of required face/face intersections depends on the number of faces in both solids:

$$i = nm,$$

where i is the number of intersections, n is the number of faces in one body, and m is the number of faces in the other body.

The number of required intersections grows rapidly (quadratically) with the complexity (number of faces) of the solids. Fortunately the different face/face intersections can be easily performed in parallel. The algorithm is structured such that it can create a cascade of *threads* (a sort of subprocess). For each pair of faces a subprocess is launched that splits itself to calculate the surface/surface intersections and the edge/surface intersections in parallel. With the availability of multiprocessor hardware the advantages of this algorithmic structure are seen as increased performance of the Boolean operations.

Imprinting and Coloring. In the intuitive approach, coloring the faces, that is, determining which pieces are inside or outside, was no problem because it could easily be seen. On the machine level other means are required.

Intersection tracks split surfaces and faces into left and right halves. Additionally, surfaces split space into halves called *half spaces*. We can classify each piece of the split face to a half space with respect to the other surface. This procedure is demonstrated in Fig. 10.

Classification is done with respect to the surface normals (colored arrows) of both surfaces (sf1 and sf2) and the intersection track.

Unusual Boolean Applications

It is easy to see that the Boolean engine is driving most machining operations. Here are some applications in which it is not so obvious.

Partial Booleans. Regular Boolean operations attempt to calculate all intersection tracks between bodies. In contrast, partial Boolean operations calculate only one intersection track. Which one depends on the particular application. One example of a partial Boolean operation in HP PE/SolidDesigner is wrapped into the extrude-to-part command. It fires a profile defined in a workplane onto a body as shown in Fig. 11. The picture shows a body and a profile set up for the

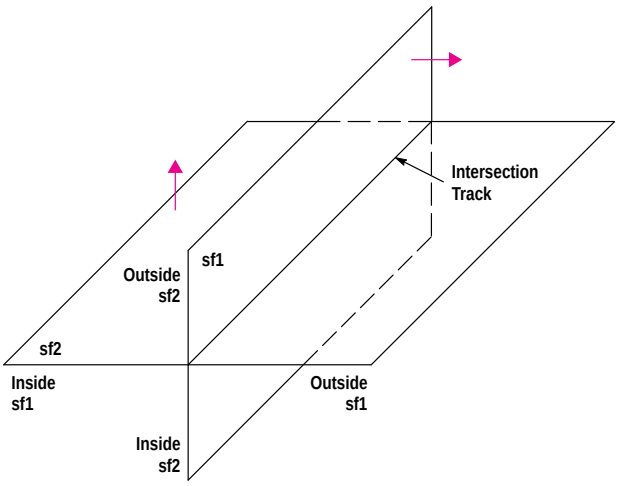


Fig. 10. Surfaces split space into halves called half spaces (inside and outside along surface normals). Each piece of a split surface can be classified as belonging to a half space with respect to the other surface.

Fighting Inaccuracies: Using Perturbation to Make Boolean Operations Robust

The robustness of Boolean operations between solids is crucial for the usability of a solid modeler like HP PE/SolidDesigner. Unfortunately, geometric modeling is like shoveling sand. With every shovel you pick up a bit of dirt. The numerically imperfect nature of geometric algorithms can challenge HP PE/SolidDesigner's Boolean engine with contradictions and inconsistencies. The Boolean engine uses a perturbation method[1,2] to push the frontier of robustness. This article explains the notion of model consistency and demonstrates what can go wrong inside a Boolean operation and what can be done to come up with a correct result anyway.

Consistency of a Solid

Looking at a solid we usually believe that it is mathematically correct, that is, that the edges are exactly on their adjacent faces and the edges meet exactly at their common vertices. In reality, however, the limited floating-point accuracy of a computer introduces errors. On the microscopic level there are gaps and holes everywhere (see Fig. 1).

The tolerable amount of error is specified by the *modeling resolution*. The system will ignore gaps and holes smaller than the resolution. However, some geometric algorithms, such as the various intersection calculations, tend to magnify errors in certain geometric configurations. This means that given an input where all errors are within limits, the result can be inconsistent in the context of the solid and prohibit the successful completion of the requested Boolean operation.

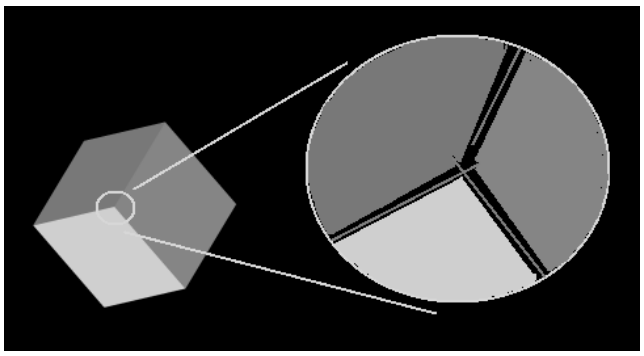


Fig. 1. In solid models edges seem to be exactly on their adjacent faces and meet exactly at their common vertices. In reality, because of the limited floating-point accuracy of a computer, on the microscopic level there are gaps and holes everywhere.

Solving the Numerical Puzzle

One area in the Boolean operation that is particularly vulnerable to numerical inconsistencies is the intersection graph construction. The graph construction assumes that all intersections of curves defined on one of two intersecting surfaces are also on the intersection track (here the term *on* means closer than the resolution). This is no problem if the surfaces are reasonably orthogonal. However, for intersections between tangential or almost tangential surfaces, a small error in the orthogonal direction of a surface implies a larger error in the direction of the surface, and this assumption becomes false.

Fig. 2 shows a shallow intersection between the two surfaces sf1 and sf2 and the intersection with sf2 of a curve (cv) contained in sf1. The curve/surface intersection point (small colored triangle) has, because of the small distance (epsilon) between cv and its containing surface sf1, moved farther away from the surface/surface intersection track (colored line) than the resolution permits. The smaller the angle β the larger the distance d from the intersection track and hence the larger the inconsistency.

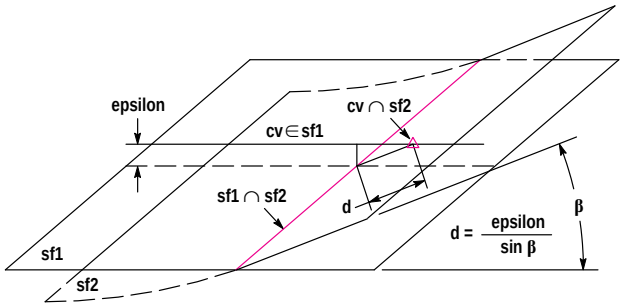


Fig. 2. A shallow intersection between the two surfaces sf1 and sf2 and the intersection with sf2 of a curve (cv) contained in sf1.

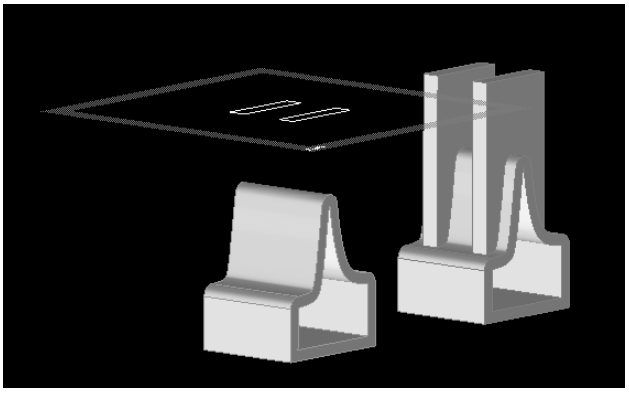


Fig. 11. A body and a profile set up for an extrude-to-part operation. To the right is the result of the operation.

extrude-to-part operation. Only the intersection graph where the extruded profile hits the body is used to build the result. To the right is the result of the operation.

Usually the extruded profile would exit the body at the bottom, producing a second intersection graph.

Reflection of Solids. Another unusual Boolean application is the reflection of solids at a plane. Fig. 12 shows a body with a green reflection plane set up. At the right is the result of the reflect operation.

This operation can be simulated with regular Boolean operations by copying, mirroring, and uniting the left body. However, this would burden the Boolean engine with difficult tangential intersections. Instead, the reflect command intersects the left body with the reflection plane to obtain an intersection graph which can be used to glue the left body

Fortunately, there is a method called perturbation than can come to the rescue in situations like this. It solves the inconsistency by moving the curve/surface intersection point along the curve until it is closer than the resolution to the surface/surface intersection track. In Fig. 2 the point will be moved to the left. When the intersection point is moved, a new error is introduced because the point is moved away from sf2. However, the overall error is reduced so that it no longer exceeds the resolution.

The perturbation method can be applied to similar situations in which even the number of intersections has to be corrected. The difference in number is a result the freedom algorithms have below the resolution. They may return *anything* in the range of the resolution.

Two Curve/Surface Intersection Points with One Surface/Surface Intersection Track. Fig. 3 shows a geometric configuration in which the intersection between sf1 and sf2 yields one intersection track (colored line) but the intersection of the curve contained in sf1 with sf2 gives two intersection points (colored triangles) which are farther than the resolution away from the track. The perturbation algorithm moves both points inwards (horizontal arrows) and contracts them into a single point (black triangle) which is closer than the resolution to the intersection track (colored line).

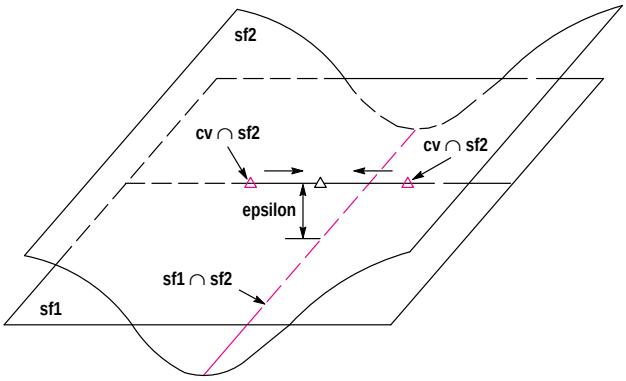


Fig. 3. A geometric configuration in which the intersection between sf1 and sf2 yields one intersection track (colored line) but the intersection of the curve contained in sf1 with sf2 gives two intersection points (colored triangles) which are farther than the resolution away from the track.

Two Surface/Surface Intersection Tracks with One Curve/Surface Intersection Point. Fig. 4 shows a geometric configuration in which the intersection between sf1 and sf2 yields two intersection tracks (colored lines) but the intersection of the curve contained in sf1 with sf2 gives one intersection point (colored triangle) which is farther than the resolution away from the tracks. The perturbation algorithm splits the single intersection into two and moves them outwards (horizontal arrows) until both are closer than the resolution to an intersection track (colored line).

References

1. H. Edelsbrunner and E. Mucke, "Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms," *Proceedings of the 4th ACM Symposium on Computational Geometry*, June 1988, pp. 118-133.
2. C. K. Yap, "A geometric consistency theorem for a symbolic perturbation theorem," *ibid*, pp. 134-142.

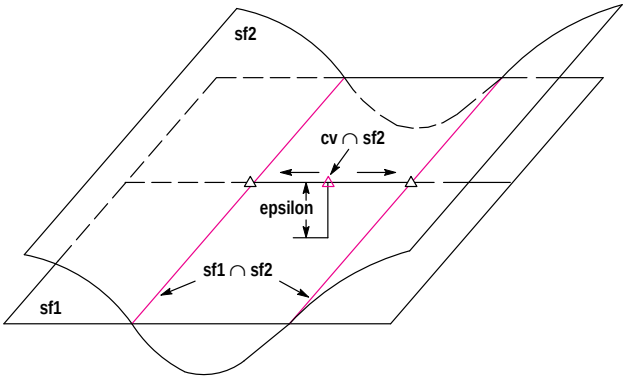


Fig. 4. A geometric configuration in which the intersection between sf1 and sf2 yields two intersection tracks (colored lines) but the intersection of the curve contained in sf1 with sf2 gives one intersection point (colored triangle) which is farther than the resolution away from the tracks.

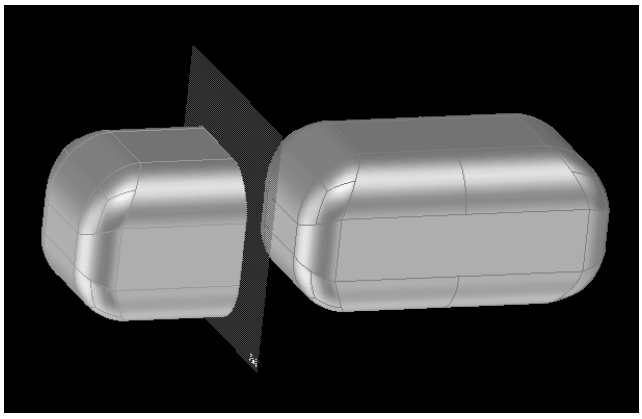


Fig. 12. A body with a green reflection plane set up and, at right, the result of the reflect operation.

and its mirrored copy together. The intersection with the mirror plane is nicely orthogonal and relatively easy to perform compared to the tangential intersections.

Acknowledgments

The development of the Boolean algorithms involved many people. Special thanks to former kernel development team members Hermann Kellerman and Steve Hull and project manager Ernst Gschwind.

Reference

1. M. Mantyla, *An Introduction to Solid Modeling*, Computer Science Press.

A Microwave Receiver for Wide-Bandwidth Signals

The HP 71910A wide-bandwidth receiver extends modular spectrum analyzer operation for more effective measurements on modern communications and radar signals.

by Robert J. Armantrout

The microwave spectrum analyzer is an invaluable instrument for making measurements on signals with frequencies ranging from 100 Hz to 110 GHz with a variety of modulation formats. The spectrum analyzer is primarily a tool for the frequency domain. The measurements for which it excels are those in which the signal parameters of interest are accessible in this domain.

For the most part, these measurements are made in a test environment, in which the signals usually originate from a signal source or from the device under test and where a physical connection is made to the spectrum analyzer with coaxial cable. In this environment, there is usually a high degree of knowledge about the signals present and the number of signals that must be characterized. Also, there is often some control over the power level of the signals present. The spectrum analyzer is normally used in swept mode. The emphasis is on the fundamental signal parameters, not on the information content present in the modulation.

Microwave spectrum analyzers are also used in the operational environment. In contrast to the test environment, the signal or signals of interest in the operational environment usually come out of the air rather than from a device under test. This means that the spectrum analyzer is connected to an antenna rather than to a device under test. Another contrast to the test environment is the number of signals present at the input to the antenna. Depending on the frequency coverage of the antenna or antennas used, the number of signals present can number in the hundreds or even thousands. In the operational environment the emphasis is on searching for signals of interest and extracting the information content of those signals. The information can have many forms including voice, video, or data. To extract this information, it is necessary to tune to the signal of interest with a bandwidth comparable to the signal's bandwidth and apply the correct demodulation.

Although the spectrum analyzer plays a major role in signal searching, it has not gained acceptance outside this role because of the limitations discussed below. Rather than the spectrum analyzer, a microwave receiver is normally used to perform the down-conversion and demodulation of wide-bandwidth microwave signals.

Bandwidth Limitations. One of the most predominant trends in modern microwave signals is the move toward wider bandwidths. This trend has been growing since the mid-1970s as satellite communications developed and radars began employing a form of spread spectrum known as chirp.

The trend continues to be evident in all areas of satellite and terrestrial microwave communications. Signal bandwidths of 30 MHz or more are typical. Furthermore, various forms of spread spectrum, such as frequency hopping or direct sequence, whether used for multipath mitigation, noise immunity, lower power density, or increased security, have led to increased bandwidths for otherwise narrowband signals.

For such signals, the bandwidth that is adequate for spectrum display or parametric measurements may not be sufficient to

Fig. 1. (left) HP 71910A receiver. (right) HP 71910A Option 011 (a display module like the one at right may be added).

preserve the information content of the signal for demodulation.

Frequency-Domain Limitations. In addition to having wider bandwidth, many modern microwave signals employ more complex modulation formats such as PSK (phase-shift keying) and QAM (quadrature amplitude modulation). Parametric measurements made in the frequency domain are not adequate to characterize these complex signals fully. Modern microwave signals can also have characteristics that vary during the sweep of a conventional microwave spectrum analyzer, making accurate characterization difficult. Finally, pulsed, bursted, gated, and time division multiplexed signals all have characteristics and information that are difficult if not impossible to extract in the frequency domain.

Amplitude-only Limitations. Because the traditional spectrum analyzer employs an envelope detector, it provides only scalar information, and phase information is lost. Since much of the information in modern complex signals is conveyed with phase shifts or variations, this limitation is significant.

Solutions

All three of the spectrum analyzer limitations mentioned above have been recognized and have led to the development of new types of instruments such as modulation-domain analyzers[1] and vector signal analyzers.[2] Although these instruments can aid greatly in the analysis of a complex signal, they do not operate at microwave frequencies and are not well-suited for direct connection to an antenna as required in an operational environment.

The HP 71910A wide-bandwidth receiver (Fig. 1) combines the attributes of a microwave receiver with the strengths of a microwave spectrum analyzer. The spectrum analyzer strengths include wide frequency coverage, synthesized 1-Hz tuning, excellent phase noise, and amplitude accuracy. The microwave receiver attributes include wider IF bandwidths and demodulation.

The HP 71910A provides easy interfacing to vector signal analyzers and modulation-domain analyzers and extends the measurement capability of these instruments into the microwave frequency range. Finally, the HP 71910A provides standard connection to commercial communications demodulator products.

Description

The HP 71910A is an MMS (Modular Measurement System) product which includes a new IF module, the HP 70911A, and a new revision of system firmware. The firmware revision permits operation of the HP 70911A with an existing microwave spectrum analyzer, the HP 71209A Option 001, and provides improved performance for signal searching (see the firmware description on page 84). The HP 70911A, which is described on page 89, provides the functions usually associated with a microwave receiver, including IF bandwidths from 10 to 100 MHz and pulse detection. The HP 70911A also offers options for FM demodulation, 70-MHz IF output, and 70-MHz channel filters. Another feature not found in other microwave receivers is the I-Q output option.

The two most common configurations of the HP 71910A operate over the frequency range of 100 Hz to 26.5 GHz. The standard HP 71910A (Fig. 1, left) provides both microwave spectrum analyzer and microwave receiver operation. An alternate configuration, Option 011 (Fig. 1, right), provides microwave receiver operation only. The rest of this article will focus on the HP 71910A Option 011 configuration.

Receiver Hardware

A block diagram of the HP 71910A Option 011 is shown in Fig. 2.The optional preamp module provides improved sensitivity and includes an internal bypass switch. The HP 70900B LO module provides the local oscillator and 300-MHz reference signals to the HP 70910A RF module. The HP 70900B also provides the firmware control of the modules that make up the HP 71910A. For operation as a spectrum analyzer or a receiver, the HP 71910A modules are slaves to the HP 70900B. The HP 70910A RF module provides microwave preselection and frequency conversion to a 321.4-MHz IF output, which provides the input to the HP 70911A module.

RF Module. The HP 70910A RF module was developed to provide wide bandwidths in the front end of the receiver. Aspects of the design important for microwave receiver operation include:

- Increased-bandwidth YTF (YIG-tuned filter) preselector
- Preselector bypass
- Mixer microcircuit for improved sensitivity
- Programmable gain at 321.4-MHz IF output.

The partial block diagram of the HP 70910A RF module in Fig. 3 shows four signal paths. The first is the low-band path, which is used for frequencies up to 2.9 GHz. There are two microwave paths, preselected or bypassed, which can operate from 2.7 GHz to 26.5 GHz. Finally, there is an IF input for use with external mixers covering from 26.5 GHz to 110 GHz (millimeter-wave frequencies).

The minimum bandwidth of the microwave preselector in previous spectrum analyzer designs ranged from 25 to 30 MHz. The design goal of the HP 70910A was to improve the minimum bandwidth of the YTF to at least 36 MHz. This was accomplished by modifying the doping profile of the YIG spheres used in the YTF. A YTF bypass path is included

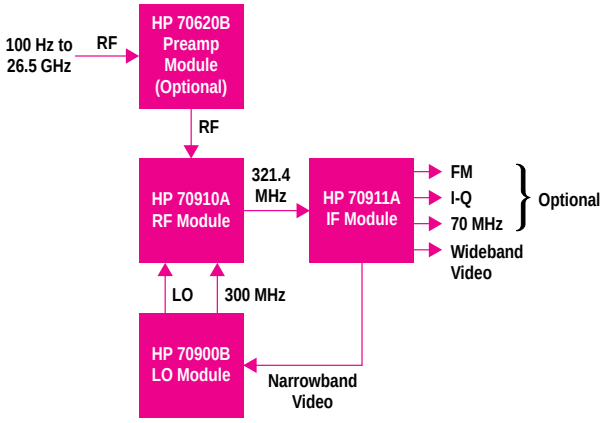


Fig. 2. A simplified block diagram of the main components of the HP 71910 Option 011 receiver.

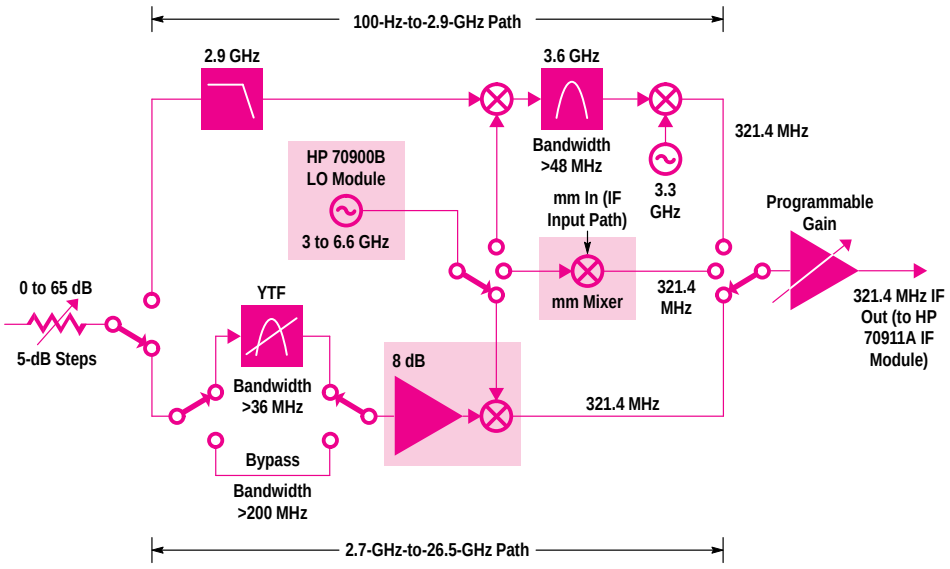


Fig. 3. A partial block diagram of the HP 70910A RF module.

to allow unpreselected operation when appropriate. When in bypass, the bandwidth of the microwave path is much wider than the bandwidth of the preselector. In addition, the group delay performance is improved when the preselector is bypassed.

The preamp-mixer microcircuit following the preselector improves sensitivity in two ways. First, the preamp compensates for the loss of the YTF while retaining acceptable intermodulation performance. Second, the mixer design takes advantage of a special diode configuration to minimize conversion loss in the harmonic-mixing bands.

The final 321.4-MHz block is the programmable-gain amplifier. The purpose of this amplifier is to maintain a constant gain from the RF input to the 321.4-MHz IF output as a function of frequency. The gain is set based on lookup table values determined during final test.

IF Module. The design goals for the HP 70911A IF module included:

- 100-MHz bandwidth variable in 10% steps
- 70-dB gain in accurate 10-dB steps
- Pulse detector for 10-ns pulses
- 70-MHz IF output
- FM demodulator.

Variable bandwidths and accurate gain are standard in spectrum analyzers, but typically at center frequencies of 3 MHz or 21.4 MHz. In the HP 70911A all variable gains and bandwidths are centered at 321.4 MHz. The higher center frequency and the higher fractional bandwidth presented significant design challenges.

An envelope detector for AM and pulse detection is also standard in spectrum analyzers, but in the HP 70911A design we had to accept a 321.4-MHz input and have bandwidth consistent with recovering 10-ns wide pulses.

Wide-bandwidth FM signals are common in both satellite and terrestrial microwave communications. For this reason, wideband FM demodulation, not found in spectrum analyzers, was an important design goal in the HP 70911A.

Within the communications industry, 70 MHz is a standard IF frequency. Most commercial communication demodulators accept 70-MHz inputs. For this reason, a 70-MHz IF output was considered essential for interfacing to demodulators for formats other than wideband AM or FM.

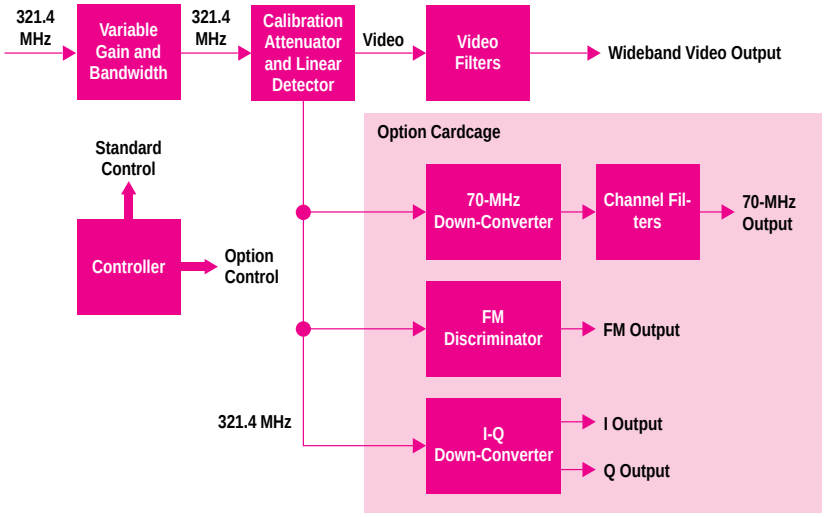


Fig. 4. The major functional blocks that make up the HP 70911A IF module.

Fig. 4 shows the major functional blocks that make up the HP 70911A. The variable gain and bandwidth block includes a bandpass filter with bandwidth that varies from 10 to 100 MHz with DAC control. The filter is a five-pole synchronously tuned design. The gain consists of seven stages of step gains interspersed with the poles of the filter.

The calibration attenuator and linear detector block includes a precision attenuator with 15-dB range and an envelope detector with 100-MHz bandwidth. The detector is followed by video gain and level control.

Several optional functions can be added for expanded receiver capability. These plug into an option card cage and are controlled over a common internal bus.

FM Discriminator. This block includes a delay line discriminator with excellent linearity and a maximum bandwidth of 40 MHz. Two sensitivity positions, 10 MHz/V and 40 MHz/V, can be selected.

70-MHz Down-Converter. This block consists of a down-conversion path and a fractional-N synthesized local oscillator. The tuning range of the LO provides, with a change of output filter, 140-MHz and 160-MHz IF outputs.

Channel Filters. The standard IF bandwidths are of the synchronously tuned class and provide very good response for pulses but lack the shape factor desired for communication signals. The channel filters provide a selection of five six-pole Chebyshev filters centered at 70 MHz for use as prefilters for the communication demodulators.

I-Q Down-Converter. This block provides I and Q baseband outputs with a 50-MHz bandwidth for each output. The local oscillator for the down-conversion is synthesized and the design is the same as that used for the 70-MHz output.

A more detailed discussion of the design and implementation of the HP 70911A is given on page 89.

Receiver Bandwidth Improvements

The increase in bandwidth for the HP 71910A is dependent on the frequency band selected. The receiver bandwidth, which depends on the bandwidth of both the RF and the IF modules, ranges from 36 MHz to 100 MHz. The RF bandwidth of the low-band path is set to 48 MHz minimum by the bandpass filter in the 3.6-GHz second IF. In the preselected microwave path, the bandwidth of the RF module ranges from 36 MHz to 60 MHz over the 2.7-GHz-to-26.5-GHz frequency range. However, when the preselector is bypassed the bandwidth of the microwave path approaches 200 MHz. Finally, when using external mixers for frequencies above 26.5 GHz, the bandwidth of the RF path will be set by the mixers, but is at least 200 MHz. The resulting receiver bandwidth for each path is summarized in Table I.

Table I
HP 71910A Receiver Bandwidths

HP 70910A Signal Path	IF Output (MHz)		
	321	140	70
Low-Band	48	48	40
Preselected	36 to 60	36 to 60	36 to 40
Unpreselected	100	70	40

Receiver Operation

To simplify the HP 71910A's operation as a microwave receiver, a personality downloadable program was created. This program, which is loaded into the HP 70900B LO module, presents the user with the display shown in Fig 5. This screen provides information to assist the user in establishing the correct gain through the receiver when other processors, instruments, or demodulators are connected at the outputs. The RF/IF Gain annotation shows the total gain from the RF input to the 70-MHz IF output. It accounts for fixed or variable gain and attenuation in both the RF and the IF modules.

In addition to calculating and displaying gain through the receiver, the receiver personality extends the gain resolution available to the user. In normal spectrum analyzer operation, the IF gain resolution is 10 dB. However, for the HP 70911A IF module, the personality combines the 10-dB resolution of the step gains with the 1-dB resolution of the internal calibration attenuator to provide 1-dB gain setting resolution over a 70-dB range.

The receiver personality also provides control of the optional receiver functions such as FM, I-Q, and channel filtering. This partitioning from the basic firmware control of the HP 70911A was made to allow for adding options in the future without the need for a firmware revision.

In addition to providing an interface for manual control, the receiver personality card also provides a programming interface for automatic operation. After the receiver personality is loaded and initialized, control extensions appear as additional programming commands not present in the basic firmware.

Microwave Vector Signal Analysis

As mentioned earlier, vector signal analyzers have baseband processing capabilities which when used with RF or microwave down-converters permit a more complete characterization of wide-bandwidth signals.

The I-Q down-conversion option of the HP 71910A was designed specifically for use with other HP digitizers and oscilloscopes. This option can also be used with a dual-channel vector signal analyzer such as the HP 89410A to

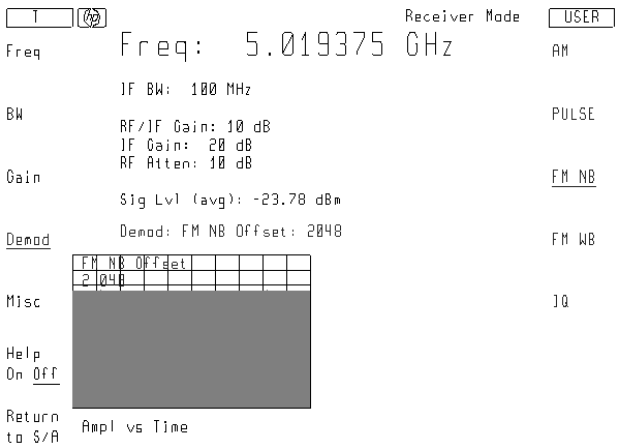


Fig. 5. The user interface screen provided by the HP 71910A receiver personality. The shaded area above is the time-domain display of an FM signal.

extend both the measurement bandwidth and the frequency range of vector signal analysis. This configuration is shown in Fig 6. A special processing mode and careful attention to calibration are required for this configuration. The HP 89410A and vector signal analysis are briefly described on page 87.

Applications

Although much of the early definition work on the HP 71910A focused on radar applications, the attention in later phases of the design shifted to microwave communications. For example, in satellite communications, which requires extensive prelaunch testing, postlaunch qualification, and periodic quality monitoring of live traffic after commissioning, the HP 71910A has much to offer.

The large investment necessary to launch a modern communications satellite makes it imperative to test the satellite and the satellite payloads thoroughly during development and manufacturing and just before launching (called high-bay testing). The number of tests required to fully characterize performance combined with aggressive launch schedules make testing throughput a major consideration.

One of the most time-intensive measurements is spurious testing. This is because of the complexity of satellites and the nature of the measurements taken using the typical spectrum analyzer approach. Sweeping a spectrum analyzer over the full transponder band with the narrow resolution bandwidth necessary for spurious testing leads to very slow sweep times and therefore very long measurement times.

Fortunately, vector signal analyzers such as the HP 89410A have much faster sweep times for the resolution bandwidths of 1 kHz or less, which are used for spurious (spur) testing. By connecting the I-Q outputs of the HP 71910A to the two input channels of the HP 89410A as shown in Fig. 6, it is possible to perform rapid spur search over a 20-MHz span. Repeating this process by step tuning the HP 71910A over all the satellite bands provides nearly a ×10 improvement in spur search speed over sweeping the spectrum analyzer with the same bandwidth over the same frequency range.

Once a satellite is commissioned and carrying live traffic, it is important to maintain the quality of the signals since failure to do so can lead to reduced revenues. One important measurement is the total power of the down link. The total

Firmware Design for Wide-Bandwidth IF Support and Improved Measurement Speed

The addition of a wideband linear IF module to a Modular Measurement System (MMS) spectrum analyzer presented two main challenges to the firmware: providing sufficient operational speed and adding new features and operations. The concern over operational speed was heightened by the fact that many of the applications targeted by this product required speed similar to that obtained by instruments that did not have to account for either software calibration or modularity.

Operational Speed

The challenges associated with operational speed involved finding a way to apply calibration in near real time and efficient handling of incoming data and temporary variables.

Calibrated Operation. To obtain calibrated data from an MMS spectrum analyzer, every trace data point obtained from the ADC must be corrected using the appropriate calibration data. This needs to be done as close to real time as possible or the lag between the incoming raw data and the completion of the correction processing will quickly become the dominant factor in the retrace dead time.

The particular calibration data that must be applied and the algorithms that must be used to apply it are dependent upon the currently active signal path of the instrument. This can change as the user selects different IF bandwidths, different ADCs, and so on. This situation is complicated further by the desire to be able to do trace math (such as calculating the difference of the active trace and a baseline trace) as the data is received. Finally, the trace data needs to be sent to the remote display (if one is active) as the processing is completed. All of these complications exist even without a linear IF module.

If a conventional program is used to apply the per-point calibration, the time to perform the necessary number of conditional tests would overwhelm the actual calculation times. An alternative approach has been used since the beginning by the MMS spectrum analyzers. Instead of performing the conditional tests for each data point, an efficient state machine constructs a program to perform the necessary calculations for the current instrument state. This is done by properly combining machine code program fragments. The construction of this program (known as the RAM program) is properly synchronized with the appropriate state changes and trace operations.

During the execution of the RAM program, calibration and interpolation table addresses and calibration constants are stored in the CPU registers whenever possible. A preloaded register set is prepared at the same time that the RAM program is constructed. If the RAM program catches up with the incoming data stream, the process running the RAM program can swap out to allow other operations to occur. By keeping all the necessary data in the CPU registers, this swapping occurs quickly.

To account for a linear IF module, various additions to the RAM program were required. Previously, all IF modules supported by the system were log IF modules. Since all data calibration occurs after the signal has traversed the IF section, it made sense to keep almost all of the correction factors in dB. This has the additional advantage of allowing simple addition and subtraction to be used to apply the calibration data. Further simplification is achieved by storing the correction factors as 16-bit, fixed-point values. A scaling factor of 100 is used. For example, a value of 10.34 dB would be stored as 1034.

With the addition of a linear IF module, the assumption of logged incoming data was no longer valid. The main alternatives were either to rework the RAM program to be able to handle linear data (including the need to do multiplication and division instead of simply addition and subtraction) or to translate the incoming linear data to log data (preinterpolation). The latter approach is much quicker at performing the calculations, but it also has the potential for a loss of accuracy. However, with the ADCs currently supported by the MMS spectrum analyzer, both the accuracy and the range are limited by the ADC, not by an initial interpolation. Thus, the preinterpolation approach was taken.

Later experimentation showed that, with the reference level set properly, a 38.5-dB range could be achieved with the required accuracy. This was sufficient for the intended use of the product. When a display of linear voltage or power is desired, a table lookup and postinterpolation is performed toward the end of the RAM program.

Hardware Caches. At fast (short) sweep times, even the RAM program, running on a 20-MHz MC68020, is not fast enough to keep up with the incoming data stream. At this point, the data is buffered for the RAM program to process when it can. At the fastest sweep times, the data acquisition loop is actually locked in

power may come from one wideband carrier or it may be the sum of hundreds of narrowband carriers. In either case, the instantaneous power over the full transponder bandwidth is desired. Using swept spectrum analyzer techniques for this measurement can limit speed and degrade accuracy. However, when the channel filters option of the HP 71910A is used with the HP 70100A power meter module, a single accurate power measurement of the full transponder bandwidth, or individual measurements of carriers with specified standard bandwidths, can be performed (see Fig. 7).

Terrestrial microwave communications is an application that involves wide-bandwidth signals with complex modulation schemes. For monitoring microwave link performance, the HP 71910A offers an optional 70-MHz IF output for connection to products that can demodulate these complex modulations. This feature might be used for assessing the BER (bit error rate) performance of the communications link. The BER performance of the HP 71910A was characterized during development. The results of those measurements are shown in Fig 8.

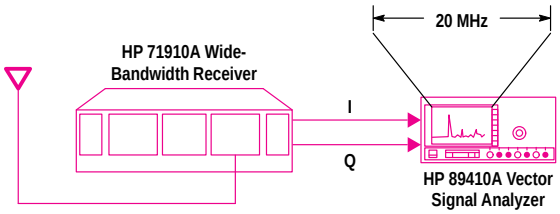


Fig. 6. The HP 71910A configured with the HP 89410A vector signal analyzer.

Other aspects of link performance are often assessed using a constellation display. The I-Q output option of the HP 71910A can be used to display the signal constellation on an oscilloscope (see Fig. 9).

Although swept spectrum analyzers have been used for characterization of radar signals for many years, the trend toward narrow pulses and intrapulse modulations have limited their usefulness primarily to spectrum displays. By connecting the HP 71910A outputs to high-speed oscilloscopes it is possible to extract much more information about the radar.

the MC68020 instruction cache to minimize memory accesses for this time-critical operation.

Software Caches. In addition to the hardware cache built into the MC68020, the MMS spectrum analyzer firmware makes use of software caches as appropriate. Because of the modular nature of the instrument, a change of state can impose a heavy calculation burden. This burden must be borne by an affordable CPU.

Detailed timing and analysis of the operation of the instrument revealed several intensive calculations that could be identified by a minimal number of internal state variables. These variables are used as tags for software caches. This approach saves 60 ms or more for some common state change operations. Use of these caches was integrated with the RAM program so that a register could access the cache data directly, avoiding costly data copying.

Further performance improvements were realized by recognizing situations in which a calculation might need to be redone because of further user inputs before a data acquisition is performed. In such cases, if it is possible, calculation is delayed.

Additional Adaptations for a Wideband Linear IF

Adding more features and operations to the MMS spectrum analyzer involved advertising the capabilities of the IF module to the analyzer and preselector centering.

Configuration Support. In addition to the changes to the RAM program, the main signal routing algorithms had to be enhanced to account for the linear IF module. In the MMS spectrum analyzer, all modules advertise their capabilities to the control module via an ASCII capability string. This machine readable string is effectively a logical block diagram of the module, including all inputs, outputs, and switching capabilities. Some of the elements of this model are named so that the control module can properly manipulate the hardware via a standardized command language.

The addition of support for a linear IF module required minimal additions to the capability string language. Most of the components of the module had already been modeled. Support for an additional value to an existing option flag was the only thing required.

Preselector Centering. The wideband IF module presented an additional difficulty with preselected systems. With a narrowband IF, the tuning of the preselector is done via peaking. In peaking a test signal is injected into the system and the preselector hardware is tuned to provide a maximum response. This approach does not work for a wideband IF module, since the peak of the passband may not be near the center. Hence, using preselector peaking with a wideband IF module could easily result in a substantially reduced available signal bandwidth.

The proper approach for adjusting a preselector to work with a wideband IF module is to center the filter based upon a user-configurable signal delta value (typically 6 dB). Centering occurs in three main stages. First, a coarse search sketches the shape of the curve and identifies where to search for the peak value. Next, a fine search identifies the actual peak. Both of these steps are similar to what occurs for preselector peaking, except that coarse values are saved. The final step involves fine searches in the areas of the curve that correspond to the user-specified delta from the peak value. In all searches, an appropriate amount of overlap is used since the curve might not be locally monotonic.

The initial implementation worked correctly, but test users sometimes complained that the preselector still wasn't being centered correctly. The typical situation was that a user had a band-limited signal path that had not been previously connected to a narrowband spectrum analyzer. Thus, the user was not aware that the signal path was the problem.

The solution to this situation is to display the centering graphically as it occurs. All of the coarse and fine points are plotted so that the user can see what is happening. In addition, the user can examine and change the selected centering setting.

Conclusion

By using the techniques described above, we were able to add support for a wideband linear IF module into the MMS spectrum analyzer family and achieve speed that matches or even exceeds that of instruments with less functionality and configurability.

Thomas A. Rice
Development Engineer
Microwave Instruments Division

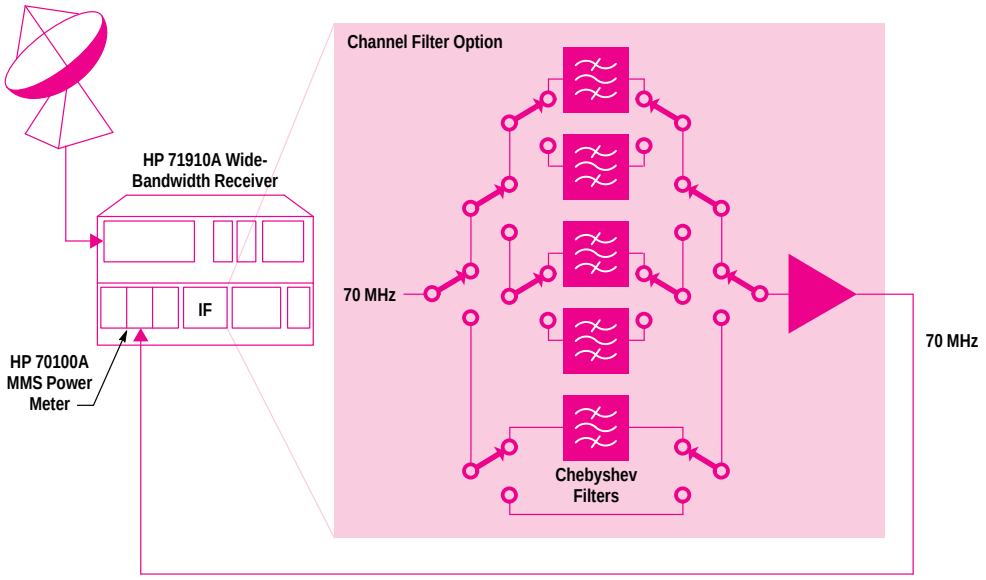


Fig. 7. The HP 71910A channel filters configured with an HP 70100A power meter to measure transponder bandwidth.

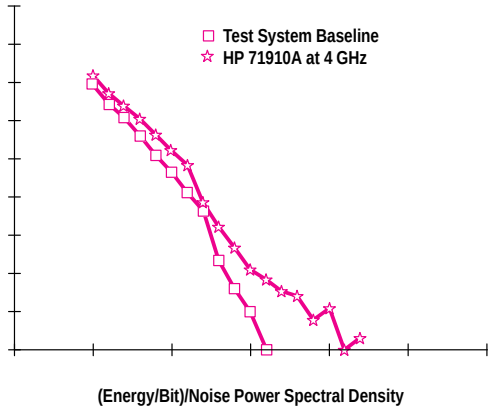


Fig. 8. Results of a 64 QAM 150-Mbit/s bit error rate (BER) test for the HP 71910A.

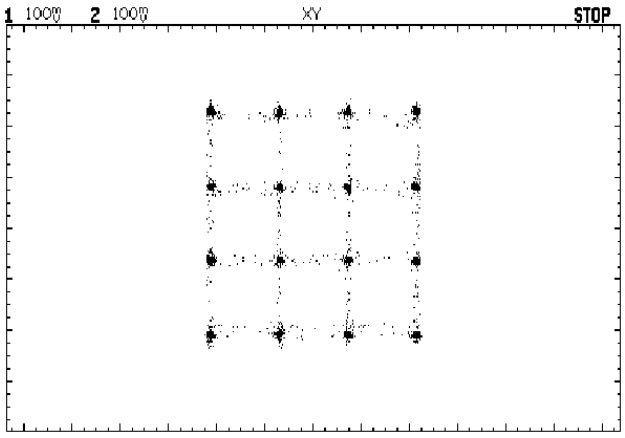


Fig. 9. A 16-QAM constellation plot from an HP 54600A oscilloscope, captured using the I-Q option of the HP 71910A.

The HP 89400 Series Vector Signal Analyzers

The HP 89400 Series vector signal analyzers are designed specifically for today's complex signals. They provide insight into a signal's time-domain, frequency-domain, and modulation-domain characteristics. The HP 89440A and HP 89441A analyzers are limited in frequency coverage to 1.8 GHz and 2.65 GHz respectively. Both are limited to a 7-MHz information bandwidth, where the information bandwidth is the widest-bandwidth signal that can be analyzed without any loss of information. The HP 71910A microwave spectrum analyzer and HP 89410A vector signal analyzer can be used together to obtain frequency coverage to microwave frequencies and information bandwidths to 20 MHz.

By itself, the HP 89410A is considered to be a two-channel baseband analyzer. Each input channel incorporates an anti-alias filter, an ADC operating at a 25.6-MHz sample rate, and dedicated hardware to perform digital signal processing. Normally, these channels are used independently. However, when used with a quadrature down-converter, such as the HP 70911A Option 004, the in-phase (I) and quadrature-phase (Q) signals from the down-converter are each connected to an input channel on the vector signal analyzer where they are digitized and then recombined into a single complex signal of the form I+jQ. Fig. 1 shows an example of the measurements obtained when the HP 89410A and HP 70911A are used together. Although the I and Q signals are each limited to 10-MHz bandwidth by the analyzer's anti-alias filters, the combined complex signal has a bandwidth of 20 MHz.

Complex Signals

In any system where the I and Q signals are analog, the accuracy of the system and its dynamic range will be limited by the orthogonality of the signals and by the match between the I and Q signal paths. Calibration routines can be used to measure and improve system performance (see Fig. 2). The system errors observed during calibration are reduced using both hardware adjustments (performed electronically) and digital signal processing techniques. Table I lists the system errors and the action taken to reduce the effects of the errors.

A program has been developed that performs the system calibration and provides some level of instrument control. This program is compatible with the HP 89410A's HP Instrument BASIC Option IC2, eliminating the need for an external controller.

Bibliography

1. *Extending Vector Signal Analysis to 26.5 GHz with 20-MHz Information Bandwidth,* Publication Number 5964-3586E, Hewlett-Packard, 1995.

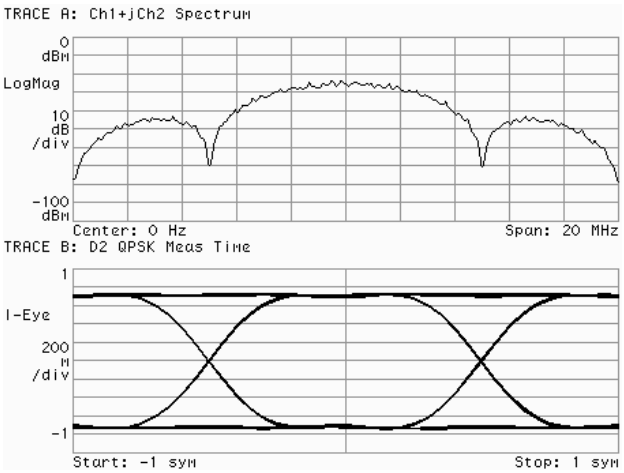


Fig. 1. The upper trace shows the spectrum of a QPSK signal operating at 10 MBits/s. The lower trace is the eye diagram obtained using the HP 89410A's optional digital demodulator.

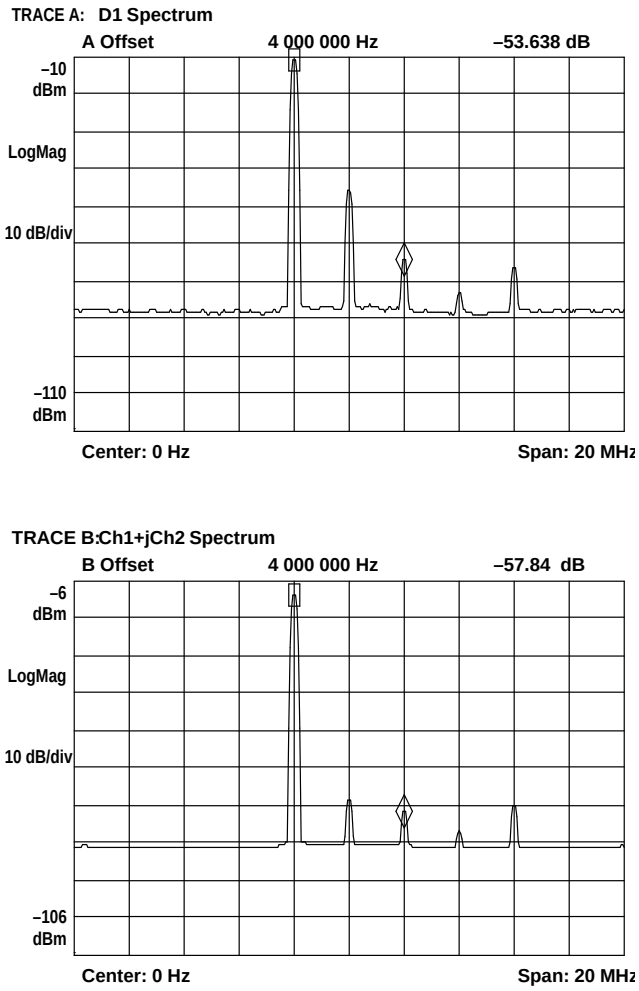


Fig. 2. The upper trace shows the spectrum computed using the I-Q signals without calibration. The lower trace is the same spectrum after calibration. Only the largest component should be present. After calibration the spectral line (center) caused by residual dc on I and Q is substantially reduced. The large spectral component has an image to the right of the center. This image, which has also been reduced in amplitude, is caused by channel mismatch.

Table I
Summary of Analyzer System Errors and Methods to Reduce Them

Source of Error	Method Used to Reduce Error: Microwave Signal Analyzer	Method Used to Reduce Error: Vector Signal Analyzer
I-Q Quadrature	Hardware Adjust	
I-Q DC Offset	Hardware Adjust	
I-Q Gain Match		Digital Signal Processing
I-Q Delay Match		Digital Signal Processing

by Robert T. Cutler
Development Engineer
Lake Stevens Instrument Division

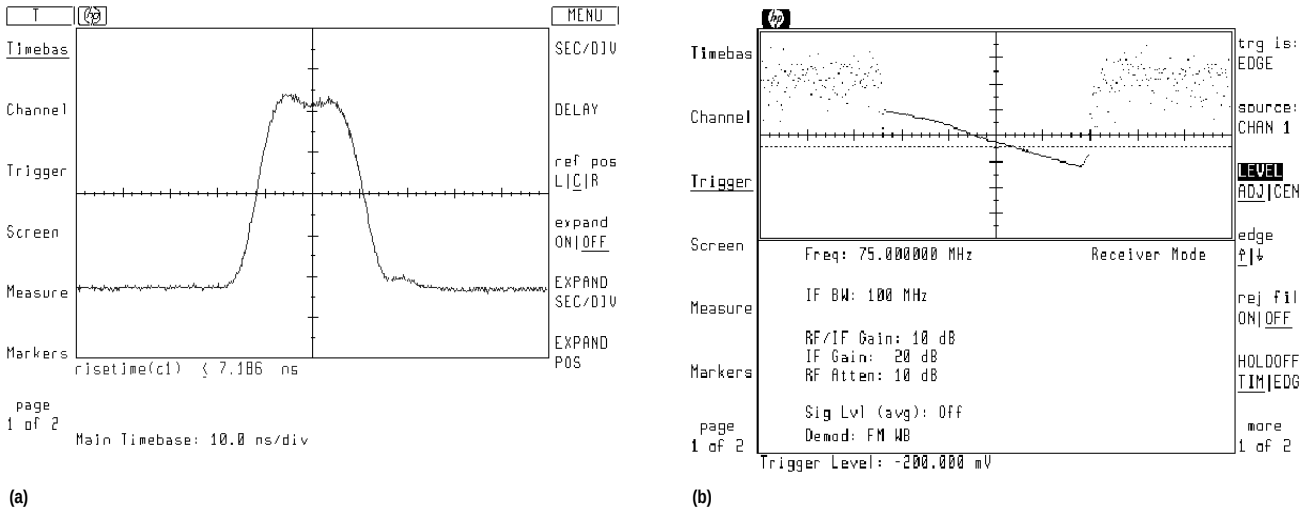


Fig. 10. Connecting the outputs of the HP 71910A to an HP 70703A high-speed oscilloscope enables the capture and display of much information from a radar signal. (a) A narrow 20-ns pulse. (b) An FM chirp.

Fig. 10a shows a narrow pulse produced by using the video output of the HP 70910A, and Fig. 10b shows an intrapulse chirp produced by using the FM output of the microwave receiver.

Acknowledgments

The author would especially like to thank John Fisher for his initial management of the project and for his support and guidance throughout. Others who made significant contributions to the project include: Ed Barich for project management and preamp-mixer microcircuit design on the HP 70910A, Mark Coomes for project management and design of the system firmware, Bill Walkowski for his efforts on market research and product definition, and Nancy McNeil who developed the receiver personality downloadable program. Finally, the author would like to thank the entire management team for their patience and support during development.

References

1. M. Wechsler, "Characterization of Time Varying Frequency Behavior Using Continuous Measurement Technology," *Hewlett-Packard Journal*, Vol. 40, no. 1, February 1989, pp. 6-12.

2. K. Burke, et al., "Vector Signal Analyzers for Difficult Measurements on Time-Varying and Complex Modulated Signals," *Hewlett-Packard Journal*, Vol. 44, no. 6, December 1993, pp. 6-30.

An IF Module for Wide-Bandwidth Signals

The HP 70911A IF module provides the HP 71910A receiver with wideband demodulation and variable bandwidths up to 100 MHz, while maintaining the gain accuracy of a spectrum analyzer.

by Robert J. Armantrout, Terrence R. Noe, Christopher E. Stewart, and Leonard M. Weber

The HP 70911A IF module provides much of the new functionality present in the HP 71910A microwave receiver. From the start, the primary design goal of the HP 70911A was to overcome the 3-MHz IF bandwidth limitation of existing Modular Measurement System (MMS) spectrum analyzers. At a minimum, we wanted a tenfold increase in bandwidth, but really hoped to achieve 100 MHz. Although bandwidth was the major design focus, several other goals were also important, including:

- Accurate gain
- Variable bandwidths
- Pulse detection
- Direct connection to demodulators
- FM demodulation
- I-Q down-conversion.

Of these goals only the first three are usually considered in spectrum analyzer IF design. The others were based on the need to better address the more complex signals employed in modern communication and radar systems.

Given the range of bandwidths required, previous spectrum analyzer IF design work has concentrated on center frequencies of 3 or 21.4 MHz. Obtaining the accuracy and stability of both gain and bandwidth required even at these IF frequencies has always been challenging. While there are a number of well-understood design alternatives and approaches available for 21.4-MHz and 3-MHz IFs, they did not exist for the 321.4-MHz center frequency chosen for the HP 70911A. Because of this some degradation of accuracy and stability was anticipated, and the design team was anxious to minimize any such degradation.

Fortunately, advances in both components and fabrication techniques were underway that were applicable to the needs of the project. The increasing availability of wide-bandwidth RF components in surface mount packages and the growing internal repertoire of surface mount manufacturing expertise suggested that the performance goals could be achieved without the need for internal microcircuit developments.

The resulting design makes extensive use of surface mount technology to achieve the goal of 100-MHz bandwidth at the 321.4-MHz center frequency while maintaining the excellent gain accuracy and stability expected of spectrum analyzers. In addition, optional down-conversion and demodulation features extend the utility for wide-bandwidth signals with complex modulations.

Fig. 1 shows the major internal functional blocks that make up the HP 70911A. A detailed discussion of the design considerations for these blocks is given below. Note that the module is partitioned into standard and option sections. An option cardcage, similar to that offered in the HP 859xE Series spectrum analyzers, provides a standard interface for all options.

Variable-Bandwidth Design

The following discussion is divided into three parts. The first part gives some background about the design of variable-bandwidth filters. The second part describes an alternative design that was considered and proven for 1-MHz-to-10-MHz bandwidths, but not included in the final product release. The final part discusses the design of the 10-MHz-to-100-MHz bandwidths of the HP 70911A.

Background. To serve as background material for describing variable-bandwidth filter design, the design approach used in the HP 70903A IF module is described here. The HP 70903A was the predecessor of the HP 70911A and used the synchronously tuned class of filters.

Synchronously tuned filters consist of several poles with the same center frequency and Q* with buffering between the stages. There are several advantages to using this particular topology, foremost being the excellent pulse response of these filters. This response allows for fast sweep speeds on a spectrum analyzer. Since we are trying to create a continuously variable bandwidth over a large adjustment range, it is also important to have a filter that can be easily adjusted. Synchronously tuned filters are easy to tune and are tolerant of a slight misalignment in different stages. Also, unlike other bandpass topologies, the Q of each stage is less than the final required filter Q.

To make these stages variable-bandwidth, a series resistance is added to reduce the Q of each of the individual stages. The individual stages look like the circuit in Fig. 2. The bandwidth of this circuit is given by the following equation:

$$BW_{stage} = (R_s + R_p)/(2\pi C R_s R_p)$$

* Here Q is filter quality factor, not quadrature as in I-Q modulation.

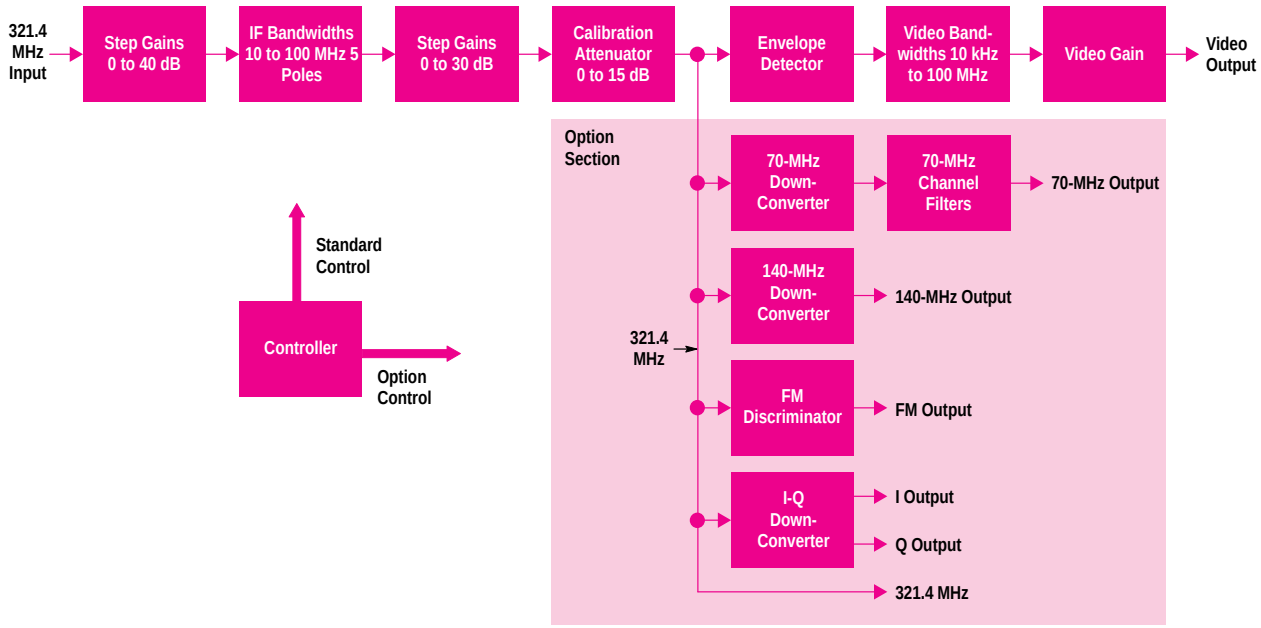


Fig. 1. Block diagram of the HP 70911A IF module.

where R_p is the equivalent parallel resistance across the tank circuit and R_s is the series Q-reducing resistance. By adjusting R_s, the bandwidth can be adjusted continuously. R_p is the combination of the input impedance of the buffer stage and the equivalent parallel resistance of the tank circuit.

By cascading several of these individual stages, a synchronously tuned filter with the desired bandwidth can be created. The equation for the bandwidth of an n-stage synchronously tuned filter is:

$$BW_{total} = BW_{stage}\sqrt{\left(2^{\frac{1}{n}} - 1\right)}$$

The typical HP spectrum analyzer has four or five stages in a synchronously tuned filter, which results in individual stage bandwidths of 2.3 to 2.6 times the overall filter bandwidth.

To implement a continuously variable synchronously tuned filter, the series resistance is created by using p-i-n diodes as variable resistors. The p-i-n diodes used are optimized as current-controlled RF resistors. The RF resistance varies with forward bias current according to the following relationship:

$$R = aI^{(-b)}$$

where a and b are constants and I is the forward bias current in the diode.

This resistance characteristic holds for frequencies above the low-frequency limit, which is set by the minority carrier lifetime of the p-i-n device. Below that frequency the devices behave like ordinary p-n junction devices and rectify the signal. This results in distortion effects that can limit the dynamic range of the filter. The recommended operating frequency is ten times the low-frequency limit, which is given by the following equation:

$$f_{min} = \frac{1}{2}\pi\tau$$

where τ is the carrier lifetime. To minimize the distortion effects from rectification, often several p-i-n diodes are used in series to minimize the signal voltage across each individual diode (see Fig. 3).

This topology depends on a low impedance driving the p-i-n diodes and a high impedance buffering the tank circuit. Typically an FET buffer amplifier is used as the amplifier at the output of each stage because of its high input impedance. Care must be taken in the design of this amplifier to avoid distortion problems caused by the large signal voltage across the tank circuit. Keeping the nonlinear junction capacitance of the FET buffer amplifier small compared to

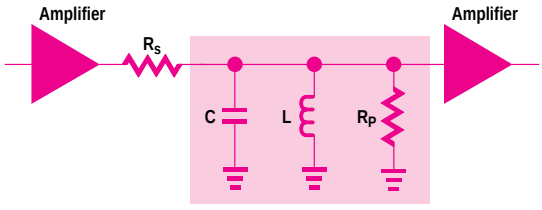


Fig. 2. RLC tank circuit with a series resistance (R_s) for adjustment. This circuit represents one stage of a synchronously tuned filter.

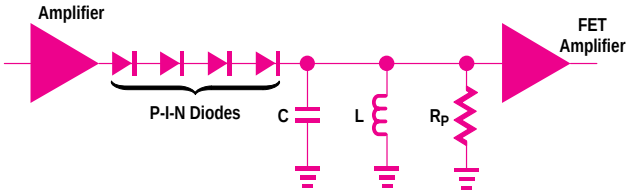


Fig. 3. RLC tank circuit with p-i-n diodes in place of a series resistance.

the capacitance of the overall tank circuit minimizes these distortion effects.

Adjusting the current in the p-i-n diodes can provide continuously variable bandwidths over a large range. Usually a digital-to-analog converter (DAC) is used to control the current in the p-i-n diodes and allow for setting different bandwidths.

This method of varying the bandwidth of the filters works very well with one slight problem. The series resistance in combination with the parallel resistance across the tank circuit creates a voltage divider. Varying the series impedance into the tank circuit not only changes the filter bandwidth, but also changes the loss through the filter as well. This amplitude change is an undesirable side effect. Several methods have been used to compensate for this change in amplitude.

One of the methods that has been patented by Hewlett-Packard uses feed-forward compensation (see Fig. 4). This method has several advantages over previous schemes that rely on feedback for amplitude compensation. The idea is to sum the proper signal at the output node to offset the drop across the series resistance element.

By summing a properly scaled version of the input signal back into the output node with a compensation resistor R_c, the voltage drop across Rs can be canceled. By setting $K = 1+R_c/R_p$, the voltage at the output node is always equal to V_{in}, independent of R_s.

Since R_p is determined by the Q of the tank circuit and the input impedance of the FET buffer amplifier, it does not vary with bandwidth. Thus R_c can be adjusted for each pole of the filter to compensate for amplitude variations. Variations in R_p over temperature can be compensated by using a thermistor in the R_c circuit to cancel their effect.

Summing a scaled version of V_{in} into the output node without introducing significant amounts of noise and distortion is accomplished in some HP IF circuits with a transformer circuit. By adding a primary winding to the tank inductor a transformer is created with a one-to-four turns ratio (Fig. 5). This sets the value of K to be four and determines the value of R_c for a given R_p as: $R_c = 3R_p$. Using a transformer with a one-to-four turns ratio yields an impedance transformation of 1 to 16. Thus, a resistor on the primary side of the transformer looks like 16 times the resistance from the secondary side. Feeding the primary side of the circuit from V_{in} through a compensation resistor requires a resistance of:

$$R_c = 3R_p/16.$$

Cascading several of these stages together implements a synchronously tuned filter that has a continuously variable bandwidth and no change in amplitude.

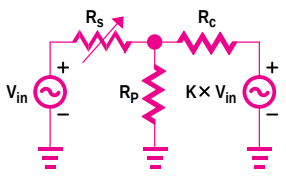


Fig. 4. Topology for a feed-forward amplitude compensation circuit.

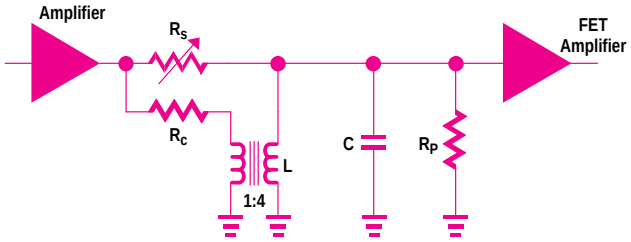


Fig. 5. Feed-forward amplitude compensated RLC tank circuit.

The HP 70903A uses four of the stages shown in Fig. 5 to implement bandpass filters with bandwidths adjustable from 100 kHz to 3 MHz at a center frequency of 21.4 MHz.

Design for 1-MHz-to-10-MHz Bandwidths. In the HP 70911A IF module we originally wanted to have continuously variable bandwidths down to 1 MHz at a center frequency of 321.4 MHz. This required an overall Q of 321.4. Even with several cascaded stages in a synchronously tuned configuration, the individual poles still required a loaded Q greater than 120. To achieve a loaded Q this high requires a device that behaves as a resonant circuit with a much higher unloaded Q. At a center frequency of 321.4 MHz there are very few choices of resonators that can achieve a Q this high. Given the size constraints of fitting on a PC board inside an MMS module, the possible solutions to this design problem were limited.

Some of the traditional choices for high-Q resonators in this frequency range include helical resonators and transmission line resonators. The size of either of these choices was the biggest obstacle to their use in the HP 70911A module. A new resonator technology was found that met all of the constraints. This resonator is a quarter-wavelength shorted coaxial transmission line formed from a high-dielectric-constant ceramic material. The dielectric constant of the ceramic is approximately 90.5, which yields a length of less than 1 inch at 321.4 MHz for a quarter-wavelength resonator. The coaxial resonators are formed with a square outer conductor 0.238 inch on a side and a circular inner conductor of 0.095-inch diameter. These dimensions are small enough to mount four of these resonators on a single printed circuit board with the appropriate circuitry to create a four-pole synchronously tuned filter. The unloaded Q of these ceramic coaxial resonators at 321.4 MHz is around 220.

A shorted transmission line (T_L) behaves like a parallel RLC resonant circuit at a center frequency corresponding to a quarter wavelength of line. An equivalent RLC lumped-element model for this circuit can be calculated by matching the slope of the reactance change with the frequency of the transmission line circuit at resonance to an equivalent RLC circuit (Fig. 6). The equivalent parallel resistance can be calculated from the Q of the resonator.

To implement a synchronously tuned filter all of the stages need to be aligned to exactly the same center frequency. By adding an adjustable capacitance in parallel with the shorted transmission line the stages can be pulled into alignment with the center frequency. This requires that the resonant frequency of the resonator be higher than the final required center frequency because the added parallel capacitance will lower the resonant frequency.

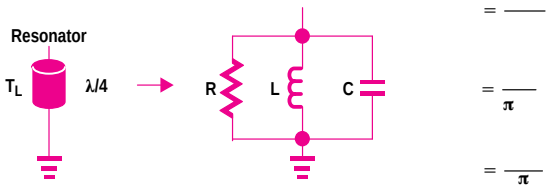


Fig. 6. Equivalent circuit for a ceramic resonator.

The resonator chosen for the HP 70911A investigation was cut to a length that corresponded to approximately 360 MHz so that it could be pulled into alignment at 321.4 MHz. Using varactor diodes for the parallel capacitance allows the alignment of all of the center frequencies using a DAC under automated computer control.

For a square transmission line with a round center conductor the characteristic impedance of the line can be approximated by the following formula:[1]

$$Z_0 \cong \frac{60}{\sqrt{\epsilon_r}} \ln\left[1.079\frac{w}{d}\right] \text{ohms}$$

where w is the width of the square transmission line, d is the diameter of the coaxial element center conductor, and ϵ_r is the relative permittivity of the dielectric.

From the dimensions given above for coaxial resonators, Z_0 is calculated to be approximately 6.3 ohms. Using the formulas given for R, L, and C in Fig. 6, the equivalent circuit of the resonator looks like Fig. 7.

To implement a four-pole synchronously tuned filter, the final Q of each stage needs to be 140 to meet the final desired bandwidth of 1 MHz. This implies a total parallel equivalent resistance of 1004 ohms. Since the resonator parallel resistance is only 1765 ohms, the total impedance of the circuit that buffers each stage must be greater than 2327 ohms. It is a challenging design task to generate a buffer stage with that high an impedance at a frequency of 321.4 MHz. To attain a maximum bandwidth of 10 MHz the equivalent parallel resistance needs to be 100.4 ohms.

The circuit topology used for the 10-MHz to 100-MHz bandwidths, which is discussed in the next section, worked well at the lower Q levels, but was unable to provide the high impedance necessary for the minimum bandwidth of 1 MHz. To attain the high impedance needed, a GaAs FET buffer stage is used across the resonator (see Fig. 8). The driver stage is a common-base configuration so the output impedance level can be set high enough to be stepped up by a tapped-capacitor transformer circuit, which is similar to the 10-MHz-to-100-MHz bandwidth circuit. The varactor diodes used to vary the capacitive taps have a tuning range of approximately 10 to 1.

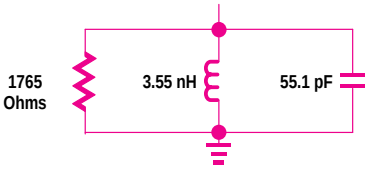


Fig. 7. Circuit values for the ceramic resonator's equivalent circuit.

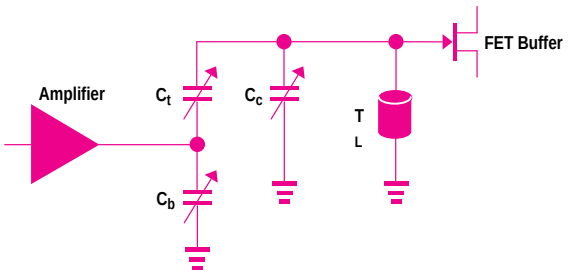


Fig. 8. Resonator with circuitry for bandwidth and center frequency tuning.

For a tapped-capacitor transformer the effective turns ratio is given by: $N = C_b/C_t + 1$. The impedance ratio varies with N^2. This impedance ratio provides the required bandwidth range but there is a drawback. The tapped-capacitor transformer also steps up the signal voltage at the input of the FET amplifier. This leads to distortion problems. The solution was to step the voltage back down with a fixed-ratio tapped-capacitor transformer (see Fig. 9). This keeps the voltage at the FET down to a level that keeps the distortion within allowable limits.

Varying C_t and C_b can set the desired bandwidth from 2.3 MHz to 23 MHz for each pole. C_c is used to adjust the center frequency to 321.4 MHz for each pole. Since the effective capacitance across the resonator changes as the tap capacitors are varied, the center frequency needs to be readjusted as the bandwidth is varied. This is accomplished with varactor diodes driven by DACs and a lookup table containing the appropriate voltage settings for each bandwidth in 10% increments over the entire range of bandwidths. Cascading four of these stages as a synchronously tuned bandpass filter yields an overall bandwidth of 1 MHz to 10 MHz.

Design for 10-MHz-to-100-MHz Bandwidths. The dynamic range limitations of the resolution bandwidth filter design approaches described above meant that they would not work for the HP 70911A. A different approach was needed. A synchronous type of filter was still desired because synchronous filters have low group delay variation. This is a requirement for good pulse fidelity, which was one of the goals for the HP 70911A. A five-resonator synchronous filter was chosen for the shape factor requirements and the range of bandwidth desired. These are two conflicting requirements because, unlike other filter types, increasing the number of resonators in a synchronous filter decreases the required Q of the individual resonators. For the required maximum

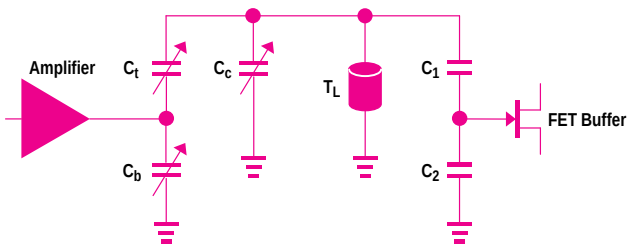


Fig. 9. 1-MHz-to-10-MHz bandwidth filter stage.

bandwidth of 100 MHz at a resonant frequency of 321.4 MHz, the fractional bandwidth of the composite filter is over 30%, and with five resonators, each tank would have a fractional bandwidth of over 80% of its center frequency.

A variable resonator with low insertion loss and low distortion was needed. Existing variable-bandwidth filters changed the Q of the resonator by varying its load resistance. For a five-resonator synchronous filter:

$$Q_{section} = Q_{overall} \times \sqrt{2^{1/5} - 1}$$

For the 10-MHz bandwidth,

$$Q_{section} = (321.4/10) \times \sqrt{2^{1/5} - 1} = 12.39$$

and for the 100-MHz bandwidth,

$$Q_{section} = (321.4/100) \times \sqrt{2^{1/5} - 1} = 1.24$$

For a single resonator, the bandwidth would be 26 MHz for the composite filter to have a 10-MHz bandwidth and 260 MHz for a 100-MHz setting. That means that a parallel resonator with an impedance of about 35 ohms at resonance would need to see a parallel resistance of between 45 and 450 ohms.

One of the ways the Q was changed in previous variable-bandwidth filters was to change the loading on the resonator with p-i-n diodes. A current source drove a series of p-i-n diodes connected to the top node of the resonator, which was connected to a high-impedance amplifier.

This is a good solution since p-i-n diodes act like inexpensive electronically controllable RF resistors. Distortion in p-i-n diodes can be reduced by putting a lot of them in series and using the same bias current. This method was tried but there was a problem. For the narrow bandwidths, a large RF voltage is present at the top node of the resonator. When this voltage is applied to the gate of a FET or the base of a bipolar junction transistor, the junction capacitance is varied by the RF voltage, causing distortion. At 21.4-MHz or 3-MHz center frequencies where this scheme has been used, the change in impedance because of this parasitic varactor is not significant. At 321.4 MHz the degradation in the third-order intercept is too great given the aggressive goals of the HP 70911A.

It seemed wise at 321.4 MHz to avoid high impedances, high RF voltages, and noise-figure-degrading p-i-n diodes. Transforming our characteristic impedance of 50 ohms up and then down using reactive transformations would allow us to avoid high-impedance amplifiers and p-i-n diodes. A capacitive transformer could be implemented with varactors to give us the desired continuous bandwidth variation. However, reference texts suggest that capacitive transformers should be used in cases where the resonators are only operated up to 20% bandwidth. In the HP 70911A, the resonators need to operate up to 81% bandwidth. It seemed like there was little hope of getting this scheme to work, but it was tried anyway.

With this topology the only place that there would be high RF voltages is at the top node of the resonator. Since there were going to be varactors at that node, there was concern about distortion. This was solved by putting the varactor diodes in a back-to-back configuration so that there would

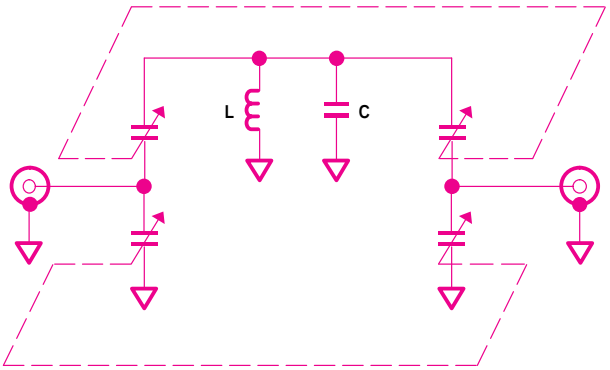


Fig. 10. Resonator for 10-MHz-to-100-MHz bandwidth. The variable capacitors are varactor diodes.

be some cancellation of the effect of the RF voltage (see Fig. 10). In this circuit, when the upper varactor increases in capacitance because of a positive swing of the RF voltage, the lower varactor decreases its capacitance, canceling out the change. Thus, the distortion problem was minimized.

The main effect of trying for over 80% bandwidth with capacitive taps is a nonideal filter shape (Fig. 11). At the wider bandwidth settings the upper tap capacitors are much larger than the lower tap capacitors. The circuit resembles a high-pass filter and doesn't have the ideal resonator rejection above resonance. This can be compensated by adding series inductors that will resonate with the upper tap capacitors (Fig. 12). The bandwidth of these outer resonators is high enough for the maximum bandwidth desired. As the main resonator bandwidth is decreased the outer resonator shifts up in frequency because the upper tap capacitance decreases. This shift does not cause trouble since the outer resonator has a bandwidth that is high regardless of the tap setting because of its 50-ohm loading on one port and variable loading on the other port.

The main resonator impedance was chosen to be 35 ohms at resonance so that for the widest bandwidths the Q-reducing resistance required was greater than 25 ohms (50 ohms at the input in parallel with 50 ohms at the output). Once that

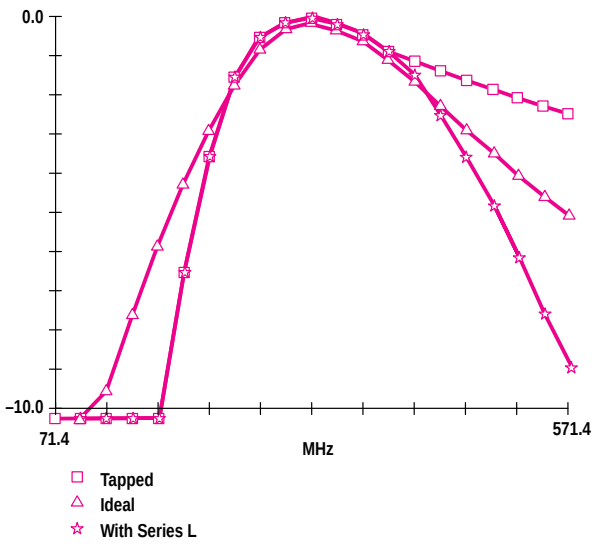


Fig. 11. The nonideal filter shape that results from using capacitance to achieve over 80% bandwidth.

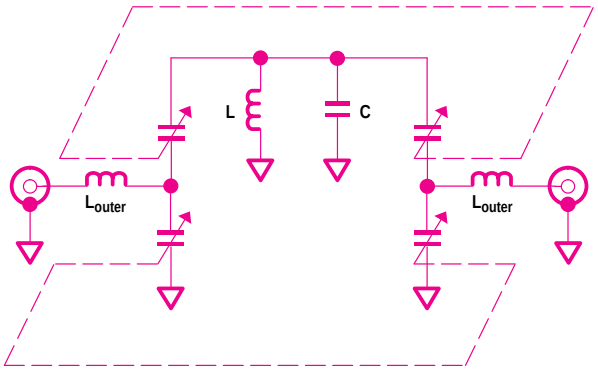


Fig. 12. Resonator compensated by adding series inductors that will resonate with the upper tap capacitors.

was decided, the values for L and C were easy to calculate. One of the complications of using the tapped capacitors is that the equivalent capacitance in shunt with the tank inductor changes with the bandwidth. This problem is solved by using DACs to control the voltages of all the varactors. A lot of calibration ROM space is required to support this circuit topology. All five resonator circuits have the lower and upper tap varactors ganged together (see Fig. 13). The shunt tank capacitors are connected to separate DAC outputs allowing independent control of the center frequency of each resonator.

The resonator circuit shown in Fig. 13 is used in the HP 70911A. The insertion loss for this circuit is less than 6 dB for the 26-MHz setting and about 1 dB at 260-MHz bandwidth. The third-order intercept point is about +29 dBm referred to the output for all settings. Group delay variation is less than half a nanosecond in wide mode and about 3 ns for the narrow-bandwidth setting.

Accurate Gain

The gain accuracy of the HP 70911A IF module depends on the gain of the seven step gains and the five filter poles and the accuracy of the calibration attenuator. How gain accuracy is achieved in each of these elements is discussed below.

Calibration Attenuator. The calibration attenuator is used during self-calibration of the HP 71910A receiver. The customer performs receiver self-calibration periodically to ensure that the receiver meets all of its specifications. This procedure measures and corrects several aspects of receiver performance. Among other things, it measures the gain of the step gain and attenuator stages and measures and corrects displayed linearity errors in the linear detector.

Since the calibration attenuator is used as a reference standard against which other parts of the receiver are measured, it is essential that the attenuator yield accurate and stable gain over the receiver's specified 0-to-55°C operating temperature range. Over this range, and over the attenuator's 0-dB-to-13-dB attenuation range, accuracy is guaranteed within 0.3 dB at 321.4 MHz.

At these frequencies, variable attenuators are traditionally designed using semiconductors with bias dependent resistivity. Examples would be p-i-n diodes with a current dependent resistance or GaAs FETs with a resistance that depends on gate voltage. Unfortunately, these types of attenuators do not demonstrate the required temperature stability. For this reason, the calibration attenuator was designed as a series of

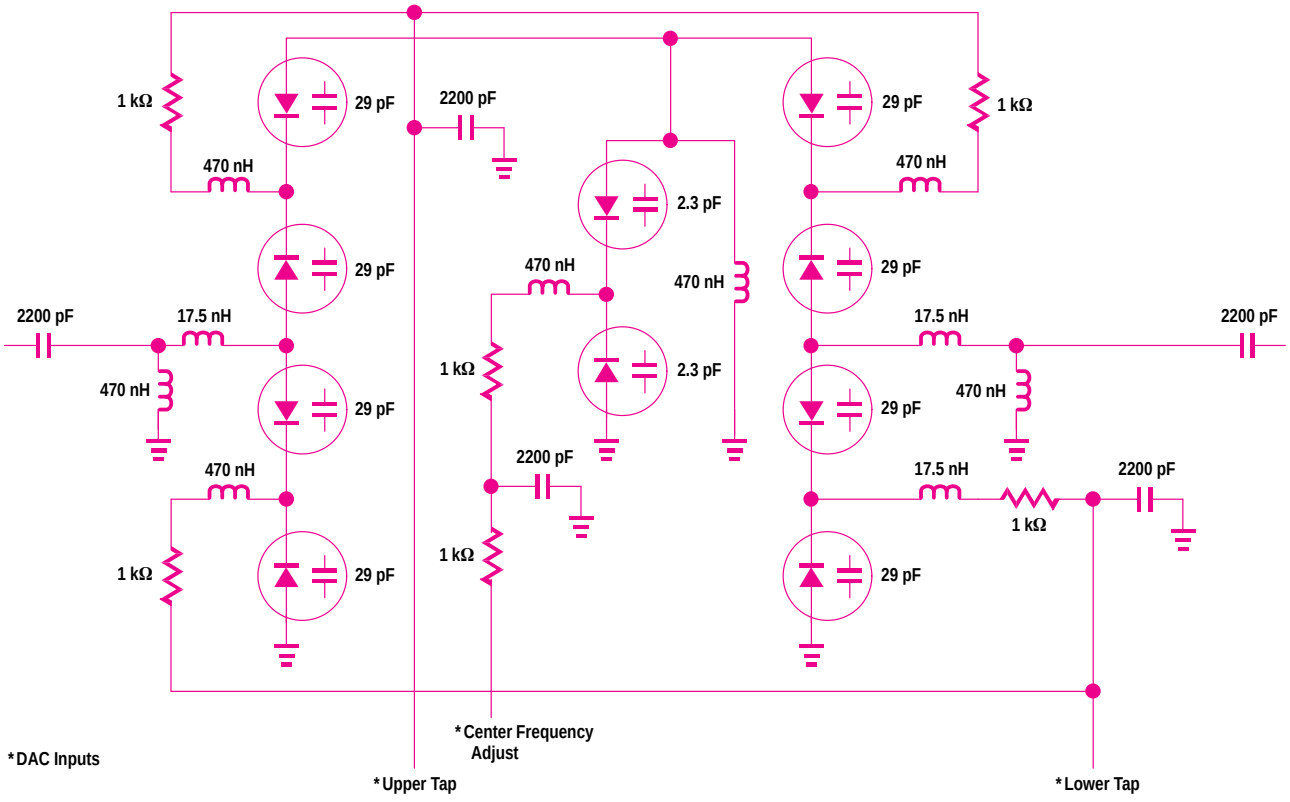


Fig. 13. Circuitry for one pole of the IF bandwidth filter showing the upper and lower tap varactors ganged together.

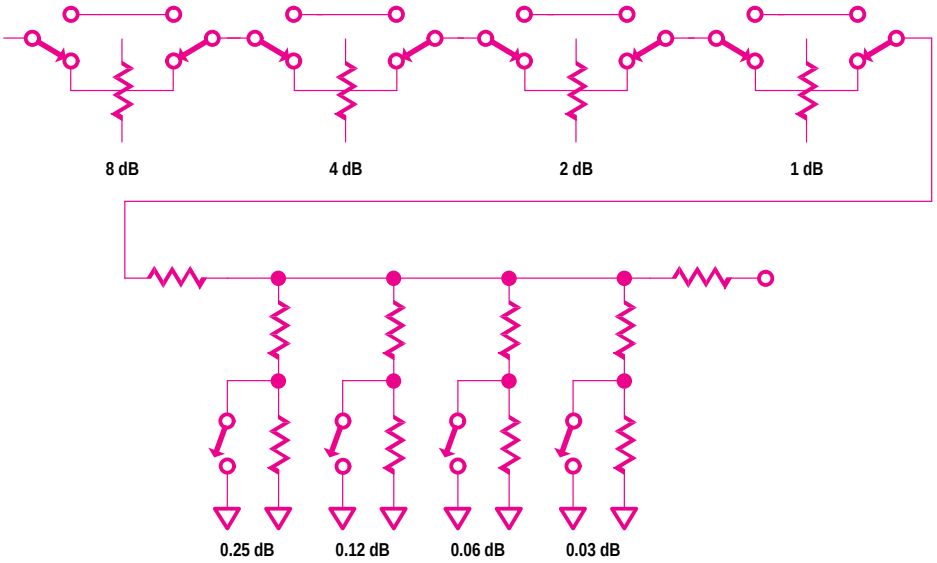


Fig. 14. The calibration attenuator is designed as a series of switchable attenuator sections.

fixed switchable attenuator sections (Fig. 14). The 1-dB through 8-dB attenuator stages are pi attenuators made with surface mount thick-film resistors. The 0.25-dB through 0.03-dB attenuator stages could not be designed as pi attenuators because the resistance values required for these very low attenuation values would not be achievable at 321.4 MHz.

Instead of trying to figure out a way to build a 0.03-dB attenuator, we built a 6-dB tee attenuator with an attenuation we could vary slightly. This was done by changing the resistance of the shunt element of the 6-dB attenuator. By switching around small resistors in series with much larger ones, very small attenuation steps can be realized. Changing only the shunt element in this attenuator does cause the attenuator's return loss to vary across its 0.5-dB attenuation range, but this effect is small enough to be acceptable.

With standard 1% tolerance resistors, the attenuation accuracy of this circuit will not be exact enough without alignment. During alignment of the HP 70911A, each 1-dB calibration attenuator step is measured and corrected to the desired value by turning on the appropriate combination of small attenuator steps. This alignment data is then stored in EEPROM.

Step Gains. The purpose of step gains is to substitute a known fixed gain ahead of the detector to enable accurate measurement of low-level signals. The ideal step gain has a 0-dB gain state and a 10-dB gain state. The implementation in the HP 70911A is shown in Fig. 15. The 0-dB (bypass) path actually has approximately 2 dB of loss, while the 10-dB (gain) path has approximately an 8-dB gain. The goal of the circuit is to make the gain difference between the 0-dB and 10-dB states exactly 10 dB. The variable attenuator in the gain path allows the gain to be trimmed to achieve this accurate gain difference. During alignment the DAC values required to trim the gain are determined for each of the step gains from measurements made at 0, 25, and 55°C. These DAC values are stored in EEPROM tables which are consulted by the module firmware during operation. As mentioned above, the calibration attenuator is used during calibration to measure the actual gain step value. In addition, because the calibration attenuator is accurate to within 0.3 dB, it can be used in conjunction with the step gain to provide accurate 1-dB gain steps over most of the 70-dB gain range.

Filter Pole Gain. As discussed above, bandwidth variation is obtained with a controlled variation of the Q of the filter pole. Because of this, the gain of the filter pole also varies with bandwidth. It is necessary to compensate for this gain variation if the module gain is to be accurate for all bandwidths. Since bandwidths are in 10% steps (10, 11, 12.1, ...), there are a finite number of bandwidths for which gain compensation is required. Associated with each filter pole is a programmable gain block (see Fig. 16). This gain block is used to provide the necessary gain compensation. The DAC values for this compensation are determined during alignment and stored in EEPROM tables, which are consulted by the firmware each time the bandwidth is changed.

In addition to controlling the nominal gain of the filter pole these programmable gain blocks also play a role in temperature compensation of the overall gain of the module. Gain drift with temperature is most troublesome during warmup. For this reason, the temperature of the module is monitored during warmup and the temperature value is used to adjust the gain to keep the output levels relatively constant. The warmup period is defined as the first hour after the module

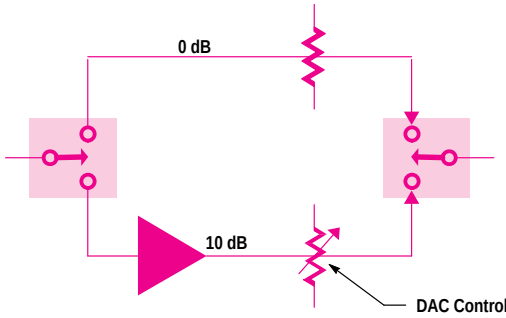


Fig. 15. A block diagram of the step gains in the HP 70911A.

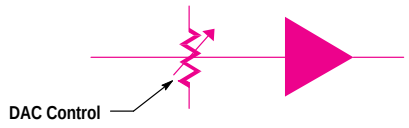


Fig. 16. A representation of a programmable gain block.

is powered on. During this period the temperature is measured once per minute and the rate of change is used to determine the size of the gain adjustment required. After the warmup period, the gain is stable for small changes in temperature so this compensation mechanism is disabled.

The module firmware orchestrates gain changes based on sampling a temperature sensor voltage with an ADC. The ADC values are used to calculate the gain change based on the following equation:

$$\frac{V_t - V_n \; G_p}{V_{55} - V_n}$$

where V_t is the voltage representation for the current temperature, V_n and V_{55} represent the voltage values for 25 and 55°C respectively, and G_p is the peak gain change between 25 and 55°C for each bandwidth. G_p is determined during alignment.

The gain change calculated is used to index into an EEPROM table to determine the DAC value necessary to achieve the desired gain. The DAC-value-versus-gain relationship is determined and stored during factory alignment.

Pulse Detection

The linear detector allows the receiver's user to recover AM and pulse modulation from the input signal. It strips the carrier from the input signal and leaves only the envelope (Fig. 17). The resulting envelope information can then be displayed on an oscilloscope, allowing the user to analyze the modulation or transient characteristics of the input signal.

The key performance specifications for the detector are bandwidth, dynamic range, and pulse fidelity. We would like the detector bandwidth to be much wider than the IF module's bandpass filters so that it does not limit the IF module's bandwidth. The bandpass filters have a maximum bandwidth of 100 MHz, which is equivalent to 50 MHz after detection. The detector is guaranteed to have at least twice this bandwidth, or 100 MHz. Dynamic range is a measure of the linearity of the detector. This is measured by changing the input RF voltage in 1-dB steps and measuring the resulting change in the dc output voltage. Ideally, it should also change by 1 dB. Our specification guarantees that over a 26-dB range, this change will be accurate within 3%.

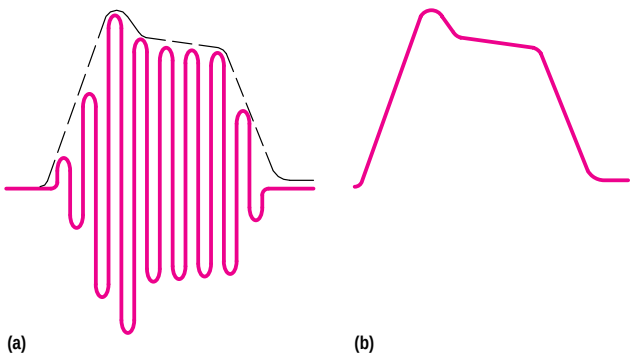


Fig. 17. (a) Input to the linear detector. (b) Output from the linear detector after the carrier is stripped off.

Previous linear detectors in HP spectrum analyzers have achieved this performance, but at much lower IF frequencies of 10.7 or 21.4 MHz. Achieving this performance at 321.4 MHz was the most challenging aspect of this design. A schematic of this circuit is shown in Fig. 18. Q1 is a common-base buffer stage that drives Q2, which is the detector transistor. Q2 and CR1 each act as half-wave rectifiers. Positive half cycles of Q1's output current flow through CR1 to ground. Negative half cycles flow through Q2's emitter and collector and develop a voltage across R1, the load resistor.

The fundamental linearity problem is that the input impedance of Q2 varies dramatically with signal level. With no input signal, Q2 is biased at 120 µA. This yields a dc resistance looking into the emitter of 217 ohms. At full-scale output, the dc emitter current is 10 mA, reducing the resistance to 2.6 ohms. This load resistance is in parallel with several parasitic loads (Fig. 19). Among these parasitic loads are Q1's output capacitance, Q1's collector bias network, the parasitic capacitance of the printed circuit board, and the capacitance of Q2's base-emitter junction. At high signal levels, Q2's input resistance is low, and essentially all of Q1's output current is delivered to the desired load. At low signal levels, Q2's input resistance is high, and the parasitic elements tend to shunt current away from the desired load. This variable current shunting degrades the linearity, so good linearity requires that these parasitic elements load the circuit as little as possible.

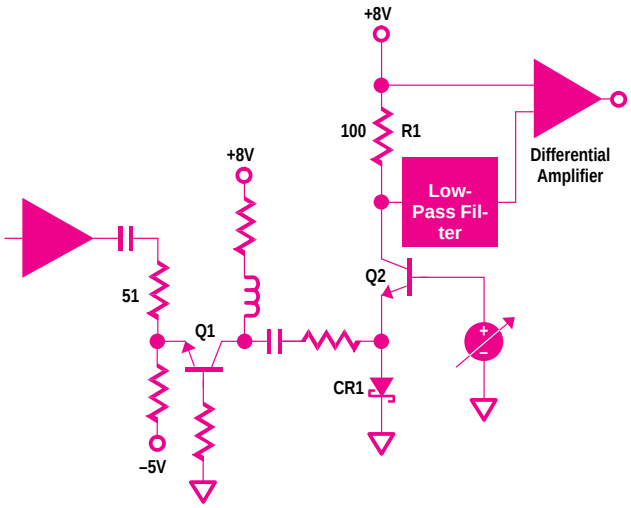


Fig. 18. Linear detector circuit.

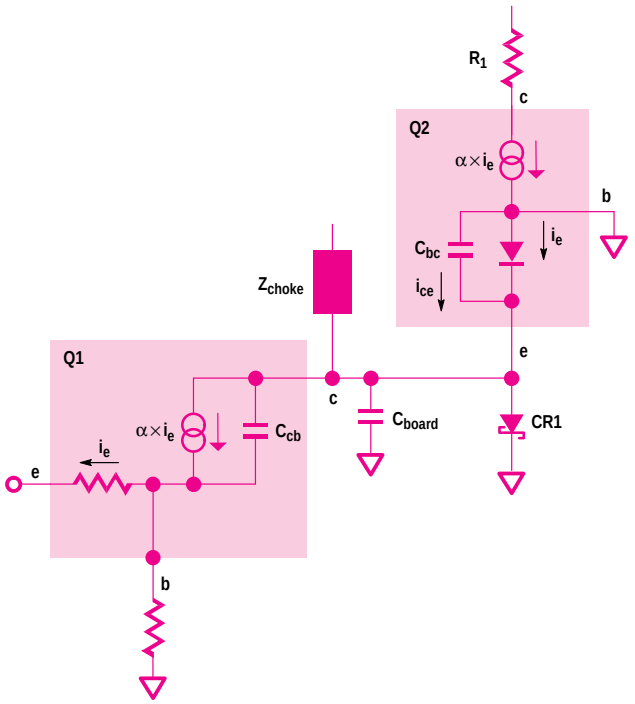


Fig. 19. Linear detector equivalent circuit.

Q1's output capacitance is minimized by using a common base configuration. Also, a microwave transistor is used because of its low capacitance. The load impedance of the collector bias network is maximized by careful design of the bias network. The printed circuit board layout is also carefully designed to make nodal capacitance as small as possible without sacrificing manufacturability.

The selection of the right transistor for Q2 was perhaps the most critical component of the design. We needed to use a microwave transistor to get the low junction capacitance we wanted. We needed two things from this transistor: a low base-emitter capacitance and, if possible, a capacitance that decreases linearly with decreasing collector current. We wanted this relationship between capacitance and current because if capacitance decreases linearly with current, then that capacitance will not degrade the detector's linearity. This is because the junction's capacitive reactance will increase as its resistance increases, and the fraction of current "stolen" by the capacitor will not vary with signal level. Since this shunting effect is independent of signal level, it will not degrade linearity.

It rarely happens, but sometimes semiconductor physics decides to give you just what you'd like. This is one of those cases. To a first-order approximation the base-emitter capacitance of a bipolar transistor is linearly proportional to bias current, at least at moderate current levels. Even better, we could easily extract this information from a transistor's data sheet curves. Low base-emitter capacitance is roughly equivalent to high f_T (transition frequency). A capacitance proportional to bias current will reveal itself as a curve of f_T that is flat versus bias current. Theory suggests, and experiment demonstrated, that the best detector transistors are those that have a high and relatively constant f_T over their entire operating current range. Unfortunately, most microwave transistor data sheets do not give f_T curves over the 100:1 range of bias currents that we wanted. Fortunately for us, we have a lot of data books and found some microwave transistors that met our needs. As expected, the transistors with the best f_T curves yielded the most linear detectors. Typical linearity error for the detector we selected is shown in Fig. 20.

The detector's output current flows across R1, generating a 1-volt drop at the maximum input level. Since the other end of R1 is tied to the 8-volt supply, it is necessary to use a differential amplifier to reference the signal to ground. A simpler approach would have been to tie R1 to ground instead of +8 volts and to tie CR1's cathode to −8 volts. This would have eliminated the need for a differential amplifier. But this would have made it difficult to achieve good pulse fidelity.

Achieving good pulse performance can be hard even with nominally linear circuits, but it is particularly difficult to do with inherently nonlinear ones like detectors. These circuits can exhibit overshoot, droop, or both on any time scale (microseconds to seconds) if their bias networks are not designed correctly. If the bias networks exhibit significant impedance at virtually any frequency below hundreds of MHz, the bias voltages in the detector can vary with the input signal, causing imperfections in the detector transient response. For this reason it seemed risky to try to build a good enough bypass network that could have presented CR1's cathode with uniformly low impedance across a broad frequency range. Rather than accept this risk, we chose to ground CR1's cathode and accept the complexity of a differential amplifier to recover the detected voltage.

The differential amplifier is integrated with a low-pass filter that removes the 321.4-MHz component from the half-wave rectified voltage across R1. This is an elliptic low-pass filter with a 200-MHz corner frequency. Even though elliptic filters have notoriously poor pulse response, we can use one here. We can do this because of the bandwidth limitation imposed on the input signal by the IF module's bandpass filters. The elliptic filter's bandwidth is four times higher than the effective postdetection bandwidth of the IF module's resolution bandwidth filters. Since these filters prevent the higher-frequency components from reaching the elliptic filter, only very low levels of ringing are observed in the detected output. We were able to demonstrate this by simulating the pulse response of the resolution bandwidth filters cascaded with an elliptic detector filter. As a result, we avoided the

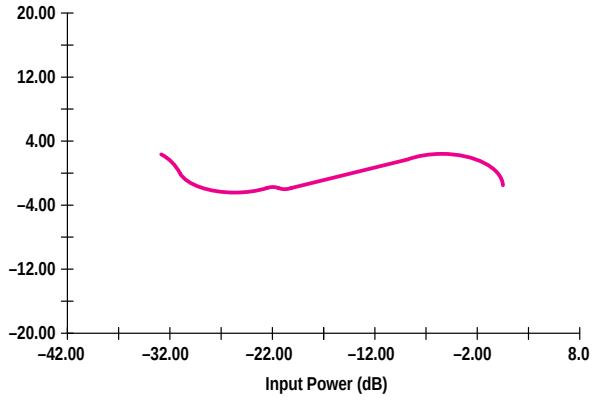


Fig. 20. Detector linearity error.

need for a more complex full-wave rectifier with its inherent carrier suppression.

Standard IF Outputs

For direct connection to commercial demodulators a 70-MHz or 140-MHz IF output is required. The HP 70911A offers either or both of these outputs as options.

A simplified block diagram for these options is shown in Fig. 21. Both down-converters use the 321.4-MHz local oscillator circuits (described later). This design has a VCO tuning range sufficient for both down-converters. The LO frequency for the 70-MHz down-converter is 391.4 MHz and the LO frequency for the 140-MHz down-converter is 461.4 MHz. These LO signals are applied to a mixer which has some buffering in front of it and is followed by an optional filter and an amplifier. Image rejection filtering is not part of the design since it is assumed that the variable-bandwidth filters are in the upstream path. The output filter is used to confine the output bandwidth to a specified amount.

The same basic design is used for both down-converters. The key difference is that in the 140-MHz design a pad follows the mixer, whereas in the 70-MHz design there is a diplexer at the mixer output, which provides a good out-of-band impedance match. The 70-MHz design has also been made available as a special option for the HP 859XE Series spectrum analyzers.

Channel Filters

The channel filters option provides an electronically switchable bank of five bandpass filters and variable gain that can be used at 70-MHz, 140-MHz, or 160-MHz center frequencies.

The input of the board goes to each filter cell through a series of GaAs switches and well-isolated stripline 50-ohm printed circuit board traces. The cells are large enough for a standard-size printed circuit board-mounted filter. The machined aluminum shield has pockets on the bottom to keep the signal pins of the filter isolated from each other. There is also a through path available for bypassing the filters. After the switching network, there is a p-i-n diode attenuator that allows continuous electronic amplitude control. Next, there is a high-dynamic-range, wide-bandwidth amplifier. The amplifier also provides temperature compensation for the gain of the board. The compensation is done by using a thermistor to vary the current in a p-i-n diode which varies the emitter degeneration impedance with temperature.

The excellent isolation, wide bandwidth, and variable gain make the channel filters a flexible option for any of the standard IF outputs.

FM Outputs

The FM discriminator generates an output voltage that is linearly proportional to the frequency of the input signal. It is used to demodulate wideband frequency modulated signals such as those found in satellite television links or chirp radars.

The key performance specification for the FM discriminator is linearity. Ideally, the frequency-input-to-voltage-output transfer function should be a straight line. Our goal was to make the maximum error from a straight line less than 1% of the full-scale output across the 40-MHz deviation range of the demodulator. The techniques used in this design were driven primarily by that goal.

Many different types of circuits have been designed to do FM demodulation. There are Foster-Seely discriminators, ratio detectors, phase-locked and frequency-locked demodulators, and slope detectors. Digital techniques, which count the zero crossings of the input signal and extract the frequency information mathematically, offer the promise of the highest linearity. These techniques are used in the HP 5371A,[2] the HP 53301A, and other modulation-domain analyzers from Hewlett-Packard. Although they achieve excellent linearity, these products are large and expensive and certainly would not fit on a single 4-inch-by-7-inch card in the HP 70911A. For these reasons, it was necessary to pursue a different approach.

Two analog demodulators seemed to offer the best potential for high linearity across a broad band: a pulse count demodulator and a time-delay discriminator. A pulse count demodulator (Fig. 22) generates a fixed-length output pulse at every zero crossing of the input signal. Since higher-frequency signals have more zero crossings, the output pulses occur more frequently. As a result, the dc average value of the output pulse train is higher for higher-frequency inputs. The low-pass filter placed after the pulse generator filters out

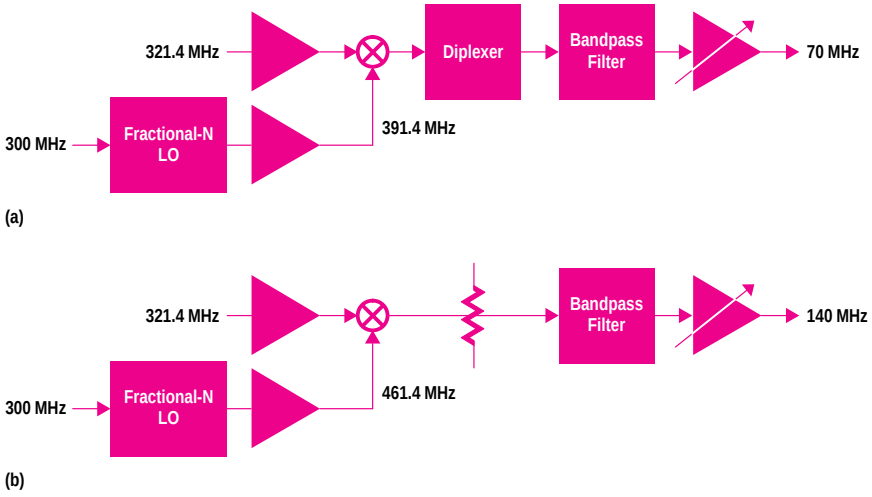


Fig. 21. Simplified block diagram of the output options for direct connection to commercial demodulators. (a) 70-MHz IF output. (b) 140-MHz IF output.

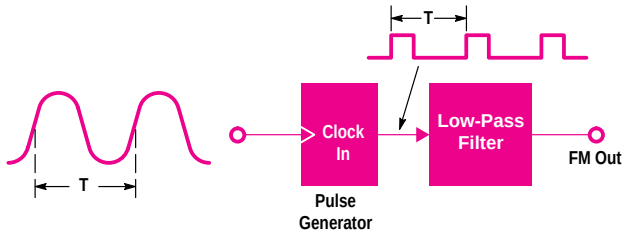


Fig. 22. Pulse count discriminator.

the carrier frequency component of the pulse train, leaving a dc value linearly proportional to the input frequency. This linear conversion of input frequency to output voltage is just what we needed to build a linear discriminator.

This type of demodulator can be implemented very simply and inexpensively by using a retriggerable one-shot timer to generate the output pulses. It does, however, have disadvantages in our application. For one, very narrow pulses would be required to make this work at 321.4 MHz. Also, these simple one-shot timers tend to have noisy outputs because of variations in the width of the output pulse.

A time-delay discriminator works by converting the input signal's frequency modulation into phase modulation (Fig. 23). A delay line delays the input signal by a fixed amount of time. A phase detector on the delay line output compares the phase of the input signal against the phase of the time-delayed version of the input. Since the phase of a high-frequency signal changes more rapidly than the phase of a low-frequency signal, the phase difference between the two inputs to the phase detector will increase linearly with frequency. The output voltage of the phase detector is proportional to this phase difference and thus, proportional to the frequency of the input signal. Typically, the length of the delay line is chosen so that the signal will be delayed 90 degrees at the center frequency of the discriminator. This gives zero volts dc output at the center frequency and centers the output in the middle of the phase detector's transfer function. This inherently linear conversion of frequency to phase seemed to make this type of circuit a logical candidate for our application.

However, this type of discriminator posed two potential disadvantages for our application. First, this discriminator is inherently limited in the maximum frequency deviation and the maximum modulation rate it can handle. Typical phase detectors only behave well when the phase difference between the inputs varies by less than 180 degrees. Because phase difference is proportional to input frequency, the maximum frequency deviation the discriminator can handle is limited. Also, the sensitivity inherently rolls off at high modulation rates. In other words, as the input frequency starts to vary more quickly, the level of the demodulated output will start to drop. The longer the delay line, the lower the modulation rate at which this will occur. In our case, we need to demodulate broad frequency deviations and as a result the maximum delay line length is limited by deviation requirements and not modulation rate needs.

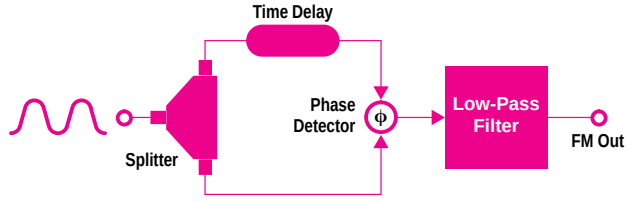


Fig. 23. Time-delay discriminator.

The second disadvantage of the time-delay discriminator is based on phase detector characteristics. Our high IF of 321.4 MHz would suggest using a double-balanced mixer as a phase detector. Conventional double-balanced mixers are designed to work with a sinusoidal RF port drive. The result is that the mixer output voltage varies sinusoidally with the phase difference between the LO and the RF waveforms (Fig. 24). Therefore, it is only linear if the phase difference between the input signals does not vary much from 90 degrees. Since we wanted good linearity, that meant a short delay line. Unfortunately, the shorter the delay line, the lower the sensitivity of the discriminator. Short delay lines mean low phase shifts and therefore low output voltages. For good signal-to-noise ratio, we wanted to maximize the time delay.

A double-balanced mixer has a sinusoidal transfer function because its RF input voltage is sinusoidal. Ideally, if its inputs are square waves, the transfer function would be linear over a 180-degree range. However, generating very fast square waves is hard, and the mixer would need a very broadband dc-coupled IF to work well. Fortunately, there is a type of double-balanced mixer that meets these requirements: the exclusive-OR gate. An ideal double-balanced mixer generates its IF by inverting the RF waveform whenever the amplitude of the LO crosses zero (Fig. 25). This is exactly what a digital exclusive-OR gate does with logic-level inputs. Thus, with this characteristic an exclusive-OR gate can be used as a double-balanced mixer.

Because of our high IF and broad frequency range, we needed to use very fast logic circuitry if we wanted this to work. Motorola's ECLinPS Lite family of emitter-coupled logic turned out to be perfect for our application. These logic gates come individually packaged in eight-pin small-outline ICs and feature rise times under 300 picoseconds. The fast, square pulses generated by this logic are perfect for making a very linear phase detector.

When logic gates are as small and fast as these, it's only natural to use them wherever you can. In the end almost all the functions on the board including limiting amplifiers, mixers,

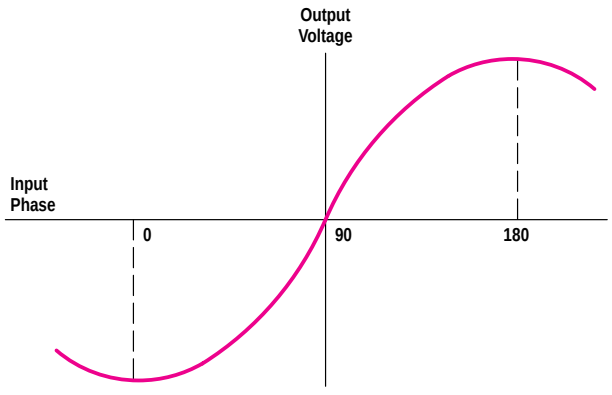


Fig. 24. Transfer function for a double-balanced mixer.

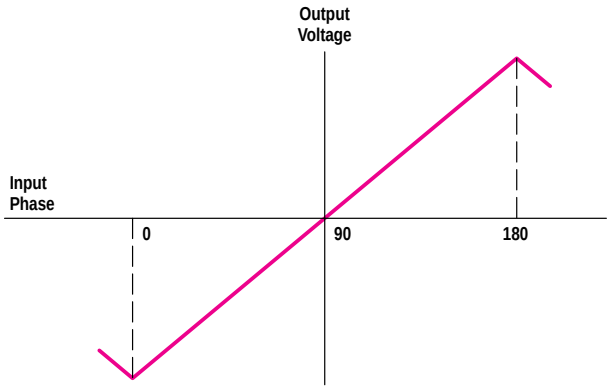


Fig. 25. Exclusive-OR transfer function.

and phase detectors were implemented using these RF logic gates. Because of the nature of FM modulation and demodulation, logic parts work well in frequency modulation applications.

Fig. 26 shows a block diagram of the FM discriminator. The 321.4-MHz input is applied to a limiting amplifier. The limiting amplifier is a high-gain stage that turns the incoming FM sine wave into a constant-level frequency modulated square wave. Given that our signal processing is done with logic parts, we obviously needed something like this to convert the input into ECL levels.

An ideal FM demodulator is insensitive to amplitude modulation of the input signal. The output voltage should not change at all when the input amplitude varies. Limiting amplifiers are used to achieve this. They have high gain and clip the level of the output signal at a predefined level. Our limiting amplifiers are implemented with ECL line receivers, which are differential-input high-gain amplifiers with ECL-level outputs. Their high gain and hard limiting allow the FM demodulator to work properly with inputs as low as −30 dBm.

The output of the limiting amplifier is a square wave with a nominal center frequency of 321.4 MHz. This is mixed against a 250-MHz LO to a lower frequency of 71.4 MHz, where the actual demodulation takes place. Originally, the intent was to do the demodulation at 321.4 MHz. As we better understood the problems we faced in trying to achieve good FM linearity, it became clear that using a lower frequency would produce better results. At a lower frequency, the period of the IF is longer. The rise time of the parts used does not change, so overall the square waves are "squarer." Our analysis of the time-delay discriminator showed that it was perfectly linear, but this is true only if the square waves are perfect.

The use of small surface mount logic parts enabled us to design compact LO generation and frequency conversion circuitry. The 250-MHz LO is derived from the 300-MHz reference frequency available in the HP 70911A. The 300-MHz signal is converted to ECL levels by a limiting amplifier. The 300-MHz reference clocks a prescaler, which divides the input frequency by six to produce a 50-MHz output. The 300-MHz and 50-MHz ECL square waves are then applied to the inputs of an ECL exclusive-OR gate. This gate performs as a double-balanced mixer, producing 250-MHz and 350-MHz outputs. The 250-MHz output is selected by a bandpass filter. This filter is ac coupled, so a limiting amplifier is placed on the output to convert the 250-MHz LO back to ECL levels.

The 250-MHz LO and the 321.4-MHz hard-limited input signal are then applied to another exclusive-OR gate. This gate is also used as a double-balanced mixer, producing outputs at 71.4 MHz and 571.4 MHz. The 71.4-Hz output is selected with a low-pass filter. The entire LO synthesis and frequency conversion circuitry occupies only 3 in^2.

The use of exclusive-OR gates and square waves, as opposed to traditional diode mixers and sine waves, has a surprising consequence. As noted earlier, a traditional diode mixer has a sinusoidal phase-to-voltage transfer characteristic. As a result, the IF out of an ideal diode ring mixer with sinusoidal inputs is another sine wave. In contrast, the logic level mixers we use here have a triangular transfer function. As a result, the IF output of these mixers is a triangular, rather than a sinusoidal waveform. In our case, we don't care whether it's sinusoidal or triangular, because we immediately convert the IF to a square wave with another limiting amplifier.

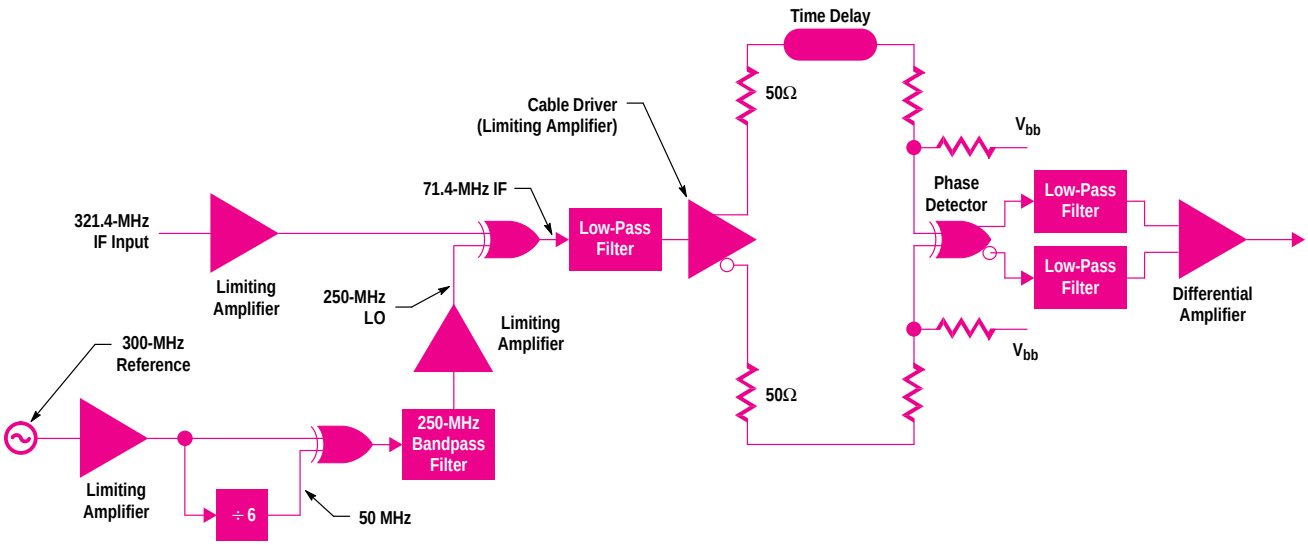


Fig. 26. FM discriminator block diagram.

The 71.4-MHz IF is next applied to the discriminator circuitry. The IF is applied to a special ECL cable driver IC which produces ECL-type outputs that have a larger than usual voltage swing. This large voltage swing allows us to place a series 50-ohm resistor on the output, cutting the signal amplitude in half. The resistor increases the output impedance of the gate to around 50 ohms, which turns out to be crucial to achieving good discriminator linearity.

The outputs of the cable driver follow two different paths. One output goes directly to the input of the phase detector. The other output goes to a delay line. This delay line is a 19-in-long 50-ohm stripline trace embedded in the middle of the printed circuit board. Numerous bends and turns squeeze it into a 1-in-by-3-in area. The board, made out of HP FR4, has a dielectric constant of about 4.5, yielding a 3.5-ns delay. This delay produces a 90-degree phase shift at the center frequency of 71.4 MHz.

The delayed and undelayed signals now meet at the phase detector, which is another exclusive-OR gate. The square waves are applied to the high-impedance input of the phase detector through a 50-ohm matching pad. The attenuation value of this pad is critical to good discriminator linearity. As mentioned earlier, good square waves are critical for good FM linearity. The attenuation value chosen strikes a balance between two "desquaring" mechanisms.

If the attenuation value is small, the delay line output will not be isolated from the 1-pF input capacitance of the phase detector. This capacitor degrades the return loss of the delay line's load. When a pulse emerges from the delay line output and hits a poor impedance match, some of the pulse's energy is reflected back into the delay line. It then travels backwards through the line to emerge at the delay line input 3.5-ns later. When the pulse reemerges here, it hits the output of the cable driver. This incident voltage disturbs the bias of the cable driver output transistors, and as a result causes disturbances in the shape of the new square wave that the cable driver is trying to generate. The degraded shape of the square wave degrades the FM discriminator's linearity. As it turns out, this effect is worst when the reflected pulse arrives at the cable driver at the edge of a new pulse. Unfortunately, this inherently occurs at the frequency where the delay line has 90 degrees of phase shift—right in the center of the passband. This effect is seen as the linearity ripple in the center of the passband (Fig. 27).

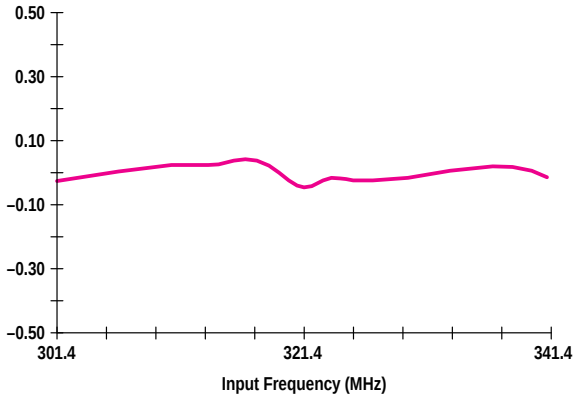


Fig. 27. Discriminator linearity error (percent of full scale).

The way to minimize this problem is to use a matching pad to isolate the delay line from the capacitance of the phase detector's input. The attenuation can only be so large because excessive attenuation introduces other problems. The attenuator reduces the voltage swing to less than ECL levels. As a result, the phase detector must provide gain to produce ECL levels at its output. Unlike the ECL line receivers, these exclusive-OR gates have a relatively low gain of 12 dB. So, with low-level inputs, the output pulses of the phase detector start to look less square. This manifests itself as the broad, slow droop in the linearity curve. In the end, an attenuation value of 4 dB was chosen as a reasonable compromise between these two linearity degrading mechanisms.

In the frequency domain, the phase detector can be thought of as producing a dc voltage proportional to the phase difference between its outputs. Looking at it in the time domain is also interesting. The two inputs to the phase detector are square waves with a fixed time delay of 3.5 ns between them. As a result, the phase detector produces output pulses of fixed 3.5-ns width. As the input frequency changes, these pulses occur more frequently, but the pulse width remains the same. This is also exactly how the pulse count demodulator works! So as it turns out, by using a linear phase detector our time delay discriminator turns out to be equivalent to a pulse count demodulator. It works as well as it does because using a delay line to fix the output pulse width is more stable than the RC time constant of a simpler implementation.

The phase detector outputs are applied to low-pass filters to remove the ac component of the pulse train. These filters have a 12-MHz bandwidth that sets the maximum frequency modulation rate the discriminator can respond to. Since the phase detector has differential outputs, a differential amplifier is used after the filters. The differential amplifier removes the dc offset inherent in the ECL level output of the phase detector. Further gain after the differential amplifier is used to give a 1-volt swing for a 40-MHz change in input frequency. The maximum frequency deviation the FM discriminator can handle is limited by the drive capability of this amplifier, rather than the discriminator circuitry itself. It has been verified experimentally that the discriminator will respond to as much as 100 MHz of deviation with essentially nondegraded linearity. A switchable amplifier provides a higher-sensitivity setting, giving a 1-volt swing for a 10-MHz frequency change.

I-Q Outputs

The I-Q down-converter (Fig. 28) recovers the in-phase and quadrature components of the input signal. The IF input, with a nominal center frequency of 321.4 MHz, is mixed against a 321.4-MHz local oscillator. This creates an IF with a nominal center frequency of zero hertz, or dc. The output bandwidth extends from −50 MHz to +50 MHz.

The input signal is split into two paths. Each of these paths goes to the RF port of a mixer. The 321.4-MHz LO is applied to the LO ports of both mixers. The LO input to one of these mixers is shifted by 90 degrees. The IF outputs are low-pass filtered to remove the image frequency, then amplified and sent to the front panel of the HP 70911A.

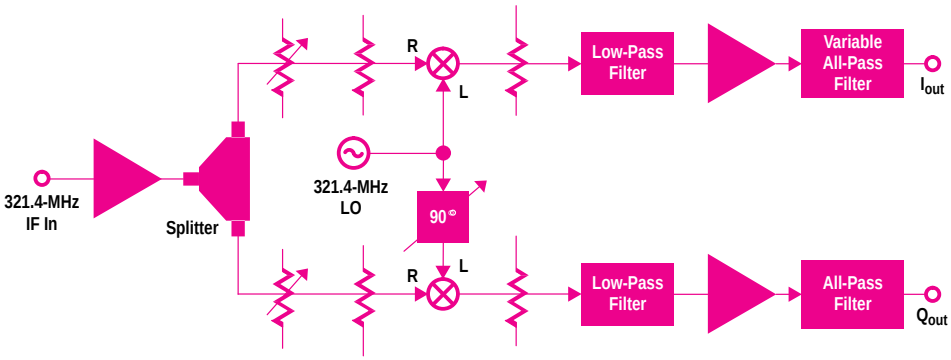


Fig. 28. I-Q demodulator block diagram.

321.4-MHz LO. The 321.4-MHz LO produces a synthesized signal that is locked to the 300-MHz reference signal and level stabilized (Fig. 29). The LO has a VCO that runs at twice the output frequency (642.8 MHz). The reason for running at this frequency is based on the availability of a 600-MHz-to-1000-MHz VCO design that has proven to have good phase noise and has been in use for some time. The VCO output is buffered and split into two paths: the main signal path and the phase-locked loop path. The phase-locked loop path goes from the splitter to a pad-amplifier-pad combination to maintain reverse isolation from the prescaler. The prescaler divides the 642.8-MHz signal by 32, 33, 36 or 37. The divide number is controlled by an HP synthesizer IC that implements the fractional-N division. The output of the synthesizer IC is equal to 300 MHz/160 = 1.875 MHz when the VCO is phase-locked. This signal goes to one input of a phase detector. The phase detector output is low-pass filtered, summed, and fed to an integrator and loop filter. This is where the synthesizer IC's noise is filtered. The noise comes from the method of fractional-N synthesis used in the IC. This noise is designed to be well outside the few kilohertz of bandwidth of the phase-locked loop where it is easy to filter.

The main signal path goes from the splitter to a divide-by-two IC. This is an ECL part that is biased in the middle of its threshold to allow for ac coupling of the 642.8-MHz VCO signal. The output of the divider is 321.4 MHz which is then input to an amplifier and resistive splitter. The splitter outputs are fed to the last gain stages of the board. These amplifiers are run well into compression to get a constant output power. The amplifier outputs are combined with a 3-dB splitter/combiner and then aggressively low-pass filtered to reject the harmonics produced by the limiting action.

I-Q Down-Converter. Two key performance specifications for an I-Q demodulator are amplitude balance and phase balance. Amplitude imbalance is the gain difference between the I and Q output ports. Ideally this gain difference should be zero across the 100-MHz input bandwidth of the demodulator. Phase imbalance is a measure of the error in the phase shift between the I and Q outputs. Ideally this phase shift should be 90 degrees across the input bandwidth.

The amplitude and phase balance of the demodulator are both factory adjusted for best performance. Variable p-i-n diode attenuators in the I and Q RF paths allow the gain of the two channels to be adjusted independently. The 90-degree phase shifter on the LO is also adjustable and is used to align quadrature. These adjustments allow us to align the channels very closely. The mixers used are purchased as a matched set, with specified gain and phase matching across our passband.

The difficult part was maintaining this balance across the entire 100-MHz input bandwidth of the demodulator. If the frequency responses of the two channels differ even slightly, amplitude and phase balance will be degraded. For this reason, we tried to make the two channels as symmetrical as possible and as flat as possible. The printed circuit board layout of the RF paths for both channels is identical so that any board parasitics will be the same for both channels. The IF circuitry was designed to be as broadband as possible. For example, the IF low-pass filters have a corner frequency of 175 MHz, even though they only need to pass frequencies as high as 50 MHz. The corner frequency was placed this high to minimize the filter's phase shift in the 50-MHz passband. Our IF amplifiers are fast op amps that provide over

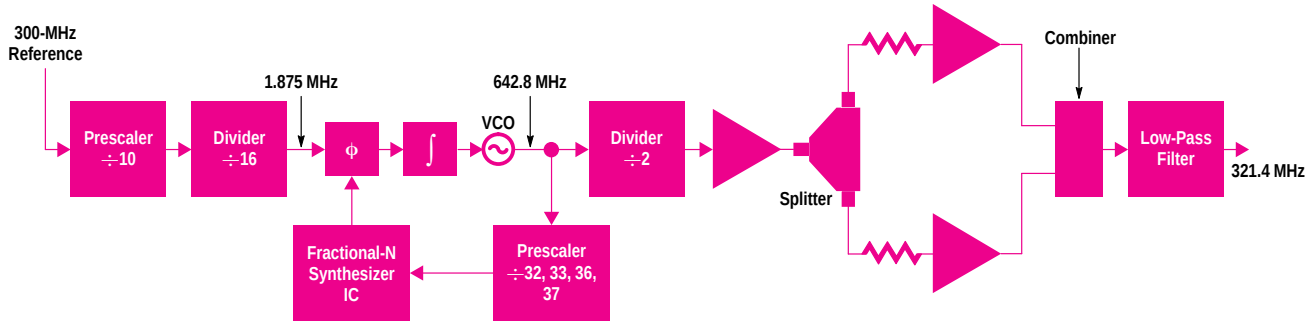


Fig. 29. Block diagram of the 321.4-MHz LO.

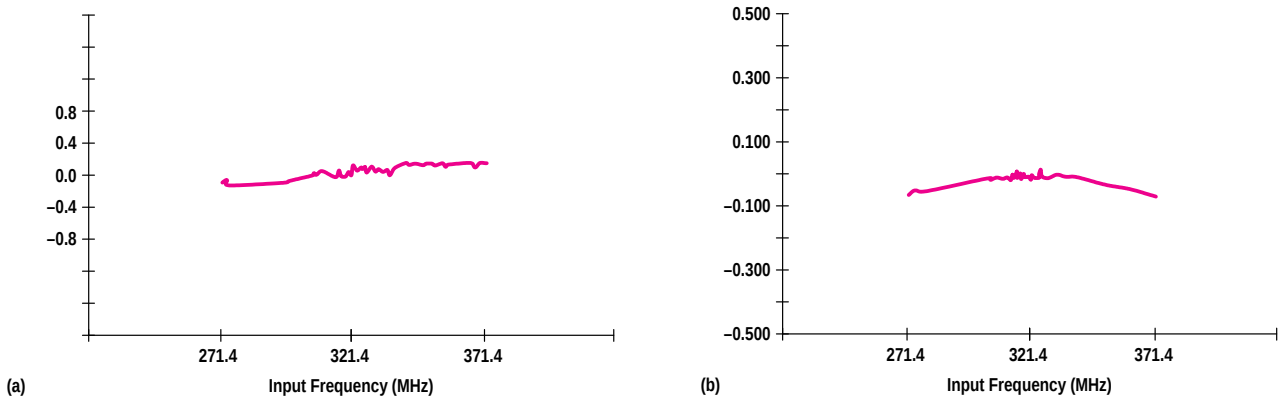


Fig. 30. (a) I-Q quadrature phase error. (b) I-Q amplitude imbalance.

200 MHz of bandwidth. These op amps are also used to minimize the phase shift in the 50-MHz passband. If these parts have significant phase shift, then there are likely to be significant phase shift differences between the two channels, and phase balance will be degraded. Representative performance for the I-Q demodulator is shown in Fig. 30.

To achieve the best phase balance across our bandwidth, an adjustable all-pass filter is used on the I-channel output. The phase shift versus frequency of this circuit is adjustable. It is used to compensate for mismatches between the channels in phase shift as a function of output frequency.

Acknowledgments

The authors would like to thank John Fisher for his support and guidance on this project. Others who made significant contributions to this project include Greg Quintana for controller and firmware design, Bill Walkowski for market research and product definition, and Gil Strand for alignment methodology and production test development. The authors would also like to thank the entire management team for their support during development.

References

1. H. Riblet, "An Accurate Approximation of the Impedance of a Circular Cylinder, Concentric with an External Square Tube," *IEEE Transactions on Microwave Theory and Techniques*, Vol. MTT-31, Oct 1993, pp. 841-844.

2. *Hewlett-Packard Journal*, February 1989, Vol. 40, no. 1, pp. 6-41.

The Log Weighted Average for Measuring Printer Throughput

The log weighted average balances the different time scales of various plots in a test suite. It prevents an overemphasis on plots that take a long time to print and allows adjustments according to the expected user profile weighting. It is based on percentage changes rather than absolute plot times.

by John J. Cassidy, Jr.

The HP DeskJet 1600C printer is designed to be used for a variety of documents, from simple memos to complex color graphics. One of the main characteristics on which the printer will be judged is throughput. We needed a way to measure throughput across a wide range of plots that would reflect a user's subjective perception of the product.

The two most common metrics—simple average and simple weighted average—had serious problems when applied to the disparate plots in our test suite. A simple and common mathematical technique was used to overcome these problems, resulting in a metric called the *log weighted average*.

This paper explains how to calculate the log weighted average, and why it is a good metric.

The Problem

We use a standard set of plots to measure the speed of the HP DeskJet 1600C printer. For the sake of this paper, I simplify the test suite down to four plots—we actually use 15. The actual timings have also been simplified and are not accurate for any version of the printer under development. The four plots are (1) text page, a normal letter or memo, (2) business graphic, some text with an embedded multicolor bar chart, (3) spreadsheet with color highlighting of some of the numbers, and (4) scanned image, a complex, full-page, 24-bit color picture.

For a given version of the HP DeskJet 1600C printer, call it version 3.0, let's say the time to process and print each of these pages is as follows:

Text page	10 seconds
Business graphic	20 seconds
Spreadsheet	45 seconds
Scanned image	10 minutes (600 seconds)

There are various things we can do to the printer to change the speed of each of these plots. Often a change will speed up one plot while slowing down another. What we need to do is compare alternative possible version 3.1s and see which one is faster overall.

Simple Average

The simple average is calculated by adding up the time for each of the plots and dividing by the number of plots. The formula for this is:

$$\text{Simple Average} = \Sigma T_i/n,$$

where n is the number of plots and T_i is the time to process plot number i.

For version 3.0 above, the sum of the four times is 675 seconds which divided by four gives a simple average of 169 seconds (rounding from 168.75).

The problem with the simple average is that it gives equal importance to each of the seconds spent on each of the plots. If a version 3.1a saved five seconds on the scanned image, this plot would go down from 600 seconds to 595 seconds and the user would barely notice. But if a version 3.1b saved 5 seconds from the text plot, this plot would go from 10 seconds to 5 seconds, twice as fast! The user would be very, very happy with the text speed.

The simple average tells me that these two changes are of equal value. So if I am using this metric, I'll go for the easy change of speeding up the scanned image by a little bit (less than 1% faster), instead of the much more difficult and more useful speedup of the text page (50% faster).

Simple Weighted Average

A common way to improve the simple average is to make use of the fact that we know how often the user is going to print each type of plot (at least we make good guesses). We know, for example, that someone in our target market will print a lot more simple text pages than complex scanned graphic pages.

The simple weighted average applies a weight to each of the plots, corresponding to the proportion of time the user will be printing that type of plot. In mathematical terms:

$$\text{Simple Weighted Average} = \frac{\sum (T_i W_i)}{\sum W_i},$$

where W_i is the weight for plot i. If the W_i add up to 1.0, the denominator can be ignored.

For the HP DeskJet 1600C printer, let's say half of the plots will be like the text page, one-fifth like the business graphic, one-fifth like the spreadsheet, and one-tenth like the scanned image. This gives the following calculation:

Plot		Time (s)	Weight	T_iW_i (s)
Text Page		10	0.5	5
Business Graphic		20	0.2	4
Spreadsheet	45	0.2	9	
Scanned Image	600	0.1	60	
Sum			1.0	78

The simple weighted average is 78 seconds.

This method of calculation is much better than the simple average. It takes into account our knowledge of the target market, and any average we come up with needs to be able to do this.

But there are still problems with this average. Say that version 3.1a speeds up the scanned image by 5% (down to 570 seconds), and version 3.1b speeds up the text page by 50% (down to 5 seconds).

We know from our own experience that speeding something up from 10 minutes to 9.5 minutes is not very significant. On the other hand, the 3.1b version, which makes the most frequent task go twice as fast, would represent a very noticeable improvement. However, the simple weighted average rates the two versions very similarly, with the 3.1a winning (at 75 s) over the 3.1b version (at 75.5 s).

Our subjective experience of time is such that we tend to notice changes not in absolute seconds, but in percentages of time. A one-percent speedup of any of the categories would be impossible to detect without a stopwatch, but a twenty-five percent speedup would be dramatic for any plot.

Criteria for a Good Average

A good averaging technique would have the following characteristics:

- It is based on percentage changes. For a short task, a small speedup is significant. For a long task like the scanned image, it takes a big speedup to make a difference. A good average would not focus on how many seconds were saved, but on what percentage of the task was saved.
- It reflects user profile weighting. For the HP DeskJet 1600C printer we need to emphasize text speed, since that is the center of our market. But for another printer aimed at another market, the spreadsheet or the scanned image might be most important. The average has to allow tailoring.
- It is invariant under a many-for-one substitution. If instead of one text page weighted at 0.5, we substituted five text pages each weighted at 0.1 into the calculation (to avoid dependence on the quirks of a single document), and if each of the five text pages took the same time as the original one (10 s) to print, the average should not change.

Log Weighted Average

The log weighted average fulfills the above criteria. Its general principle is to use a standard mathematical technique (logarithms) for keeping large and small numbers on the same scale.

The formula for the log weighted average is:

$$\text{Log Weighted Average} = \exp\left(\frac{\sum (\ln T_i)\, W_i}{\sum W_i}\right),$$

where ln is the natural logarithm (log to the base e), and exp is the exponent function, e to the x. As before, if the sum of the weights is 1.0,

$$\text{Log Weighted Average} = \exp\left(\sum (\ln T_i)\, W_i\right).$$

For our example, the calculation would be:

Plot	T_i (s)	$\ln T_i$	Weight	$(\ln T_i)W_i$
Text Page	10	2.30	0.5	1.15
Business Graphic	20	3.00	0.2	0.60
Spreadsheet	45	3.81	0.2	0.76
Scanned Image	600	6.40	0.1	0.64
Sum				3.15

$$\text{Log Weighted Average} = e^{3.15} = 23.4\ \text{s}$$

One of the first things you notice about the log weighted average (aside from the fact that it took an extra step to do the calculation) is that the result of 23 seconds is shorter than the results of the other two calculations. The simple average gave 169 seconds, and the simple weighted average gave 78 seconds. This is because the more sophisticated averages do a progressively better job of moderating the influence of the very long 10-minute scanned image plot. Also, this example was artificially constructed with a wide variation in plot times. Often we deal with plots that are more similar than these. If the plots were very similar and every plot in the test suite had exactly the same timing, say 30 seconds, then it wouldn't matter which method you used. All three methods would give the same average: 30 seconds.

Rule of Thumb

The biggest drawback of the log weighted average is that it is less intuitive than the other two methods. There is something basically counterintuitive about using logarithms if you aren't a professional mathematician. They tend to throw off our mental approximations of what is reasonable.

However, there is a relatively simple rule of thumb to help us know what to expect when doing comparisons: *A small percentage change in one component is equivalent to the same percentage change in another component, multiplied by the ratio between their weights.*

In our example, this means that a small change in the text page (with a weight of 0.5) would be five times as important as a change in the scanned image (with a weight of 0.1), and two and a half times as important as a change in the spreadsheet or business graphic (with a weight of 0.2). Thus, we would expect a 1% change in the text page to be equivalent

to a 5% change in the scanned image or a 2.5% change in the other two plots.

This approximation is very close. A 1% speedup in the text page, from 10 s to 9.9 s, reduces the overall log weighted average from 23.4 to 23.3 seconds. The equivalent change required for one of the other plots to get the average down to 23.3 is shown in Table I.

Table I
Equivalent Speedups (Small Deltas)

Text page	10	s →	9.9	s	=	1.00%	faster
Business Graphic	20	s →	19.5	s	=	2.48%	faster
Spreadsheet	45	s →	43.9	s	=	2.48%	faster
Scanned image	600	s →	571	s	=	4.90%	faster

As changes get bigger, the rule of thumb becomes less accurate. If you make a big change in one of the components, like speeding up the scanned image by 40%, you stray farther from the expected equivalent speedups of 20% (half as much) for the spreadsheet and business graphic, or 8% (one fifth as much) for the text page. This change brings the log weighted average down to 22.2 seconds. Table II shows the equivalent speedups for larger changes.

Table II
Equivalent Speedups (Larger Deltas)

Text page	10	s →	9.03	s	=	9.7%	faster
Business Graphic	20	s →	15.5	s	=	22.5%	faster
Spreadsheet	45	s →	34.9	s	=	22.5%	faster
Scanned image	600	s →	360	s	=	40.0%	faster

The Exact Rule

Exact calculation of equivalent speedups for any situation using the log weighted average can be done using the following rule: *Multiplying the time for component A by a factor r is equivalent to multiplying component B by r raised to the power W_A/W_B, the ratio of the weights of the two components.*

For example, if we multiply the text page time by 1.2 (slowing it down by two seconds), that would raise the log weighted average from 23.4 seconds to 25.6 seconds. To get an equivalent change by altering the scanned image time, we would have to multiply it by 1.2 to the fifth power (the ratio of the text page weight to the scanned image weight is five), or $600 \times 1.2^5 = 1493$. Thus, by changing the scanned image time to 1493 seconds, we could also raise the average from 23.4 to 25.6 seconds.

For very large changes in any of the components, the log weighted average gives results that can conflict with intuition. For example, speeding up the text page from ten seconds to one second would improve the average dramatically. Such a speedup is wildly improbable for the HP DeskJet 1600C printer, but can be anticipated for some comparable printer to be developed in our lifetime.

To get an equivalent improvement in the average by only changing the scanned image, we would have to print it in $600 \times 0.1^5 = 0.006$ second (which probably violates some laws of physics). You can use the exact rule to verify that the same sort of numerical blowup results when you try to compare any two printers that are greatly dissimilar. This is not a particular problem for us. Greatly dissimilar printers also have dissimilar weighting profiles, and we don't know any way to compare them well.

Usefulness of the Log Weighted Average

The log weighted average is designed around a user's subjective perception of printer speed. It assumes the common situation in which a user is working at a computer, sends something to the printer, and somehow notices how long it takes to come out. There is also an assumption that if something takes twice as long, the user is unhappy and if something takes half as long, the user is happy, and the unhappiness in the first situation is roughly equivalent in intensity to the happiness in the second situation.

There are some situations for which this isn't true and the log weighted average is the wrong average to use. For example, you could have a printer in continuous use with no stopping except to add paper and change pens. This might be at a real estate office producing a large number of personalized letters and envelopes each day and a smaller number of scanned house photos. For a customer like this, the subjective perception of speed is not important. Two seconds saved on a text page is no more important than two seconds saved on a scanned image. The simple weighted average would be the correct average to use here.

Our success with this technique resulted from regular application. On the HP Deskjet 1600C project, we timed the 15-plot test suite twice a month. This helped us quickly identify and resolve issues that might otherwise have caused problems.

Conclusion

The log weighted average does a good job of balancing the different time scales of various plots in a test suite. It prevents an overemphasis on plots that take a long time to print, and allows adjustments according to the expected user profile.

The main cost of the log weighted average is that it is less intuitive than other methods. The rule of thumb and the exact rule are good guides as to how the average will react.

The log weighted average has limits, but for comparing two reasonably similar printers in a normal home or office environment, it gives extremely helpful results.

Acknowledgment

Thanks to Jeff Best of the San Diego Printer Division for his comments and discussion.