EFFICIENT INTERSECTION TESTS FOR OBJECTS DEFINED CONSTRUCTIVELY

by

Stephen Cameron

Technical Monograph PRG-85
ISBN 0-902928-64-3

July 1990

Oxford University Computing Laboratory
Programming Research Group
11 Keble Road
Oxford OX1 3QD
England

Oxford University Computing Laboratory
Programming Research Gronp
11 Keble Road
Oxford OX1 3QD
England

Efficient Intersection Tests for Objects Defined Constructively*

Stephen Cameron

Abstract

Testing for the existence of intersections is an important part of algorithms for interference detection, collision detection, and the like. We describe three techniques that can be used to implement an efficient intersection detection routine when entities are described *constructively*; that is, as set combinations of primitive entities. All three techniques are described in a domain where constructive solid geometry is the principal entity description used, although their use in boundary representation schemes are also discussed. The first technique, called S-bounds, is a method of reasoning about where intersections may be taking place; in practise it is fast, and often sufficient. S-bounds can also be used as a general constraint manipulation method over Boolean algebras. The second technique is based on spatial subdivision, and is used mainly to improve the speed of the intersection test. The third technique is employed only on the regions of space that are left by the first two techniques; it is a specialisation of the "classical" technique of generate-and-test. The combination of these techniques has been implemented as an intersection detection routine which shows a speedup over the "classical" algorithm of about two orders of magnitude.

*This paper is a slightly longer version of one that appeared in the *International Journal of Robotics Research* 8(1)3–25, February 1989

1 Intersection Detection Problems

The general intersection detection problem can be formulated as: 'Given two subsets A and B of $\Re^n$, do they have any point in common?' A common use of intersection detection is to solve the *interference detection* problem, viz.: 'Given two objects, O_A and O_B, do they interfere?' This can be solved by considering the point sets A and B, which consist of all the points in the objects O_A and O_B, and performing an intersection test in $\Re^3$. Collision detection can also be formulated as an intersection test, this time in $\Re^4$ [Cam84]. In fact, many of the algorithms used in solid geometry, including many of those useful to roboticists, rely heavily on being able to perform intersection tests between entities of various dimensionality—for example, ray tracing considers the intersections of a line with a solid.

Entities are often described to a computer constructively; that is, as set combinations of simpler entities. In such systems it is possible to reformulate an intersection test as one of *null object detection* (NOD): given the two entities A and B, test whether their set intersection $A \cap B$ is the null set. In the rest of this paper we will concentrate on the solution of NOD in the context of a constructive solid geometry (CSG) modelling system; that is, a system in which objects are described to a computer as set combinations of primitive objects (such as parameterised cuboids and cylinders) and which keeps these description internally as its principal representation of the objects. However, the techniques described here can also be of use in other systems, such as some boundary representation (B-rep) systems which also keep a record of the steps used to construct the objects, and the modifications required will be outlined as appropriate. (For a discussion of the various types of solid modelling systems available, see [RV82, RV83].)

The algorithm to be described consists of three separable parts. The first part is based on a new method for reasoning about the parts of space that could be occupied by the set we are testing for nullity, called the *S-bounds* method, and because this method is new its description occupies much of this paper. The use of this method is sometimes enough in itself to partially decide the NOD problem, as it can give the answer "definitely null". (As an example, the two robots in figure 1 are shown to be not intersecting just by using the S-bound method.) The second method is based on a spatial subdivision technique used in computer graphics; this is also a partial decision procedure, with possible answers "definitely null", "definitely not null", and "don't know". Both of these two techniques also *localise* the problem; that is, reduce the 'volume' of 'space' in which it is necessary to search for evidence of non-nullity. The third method is the exhaustive method that is used when all else fails, and is based on the

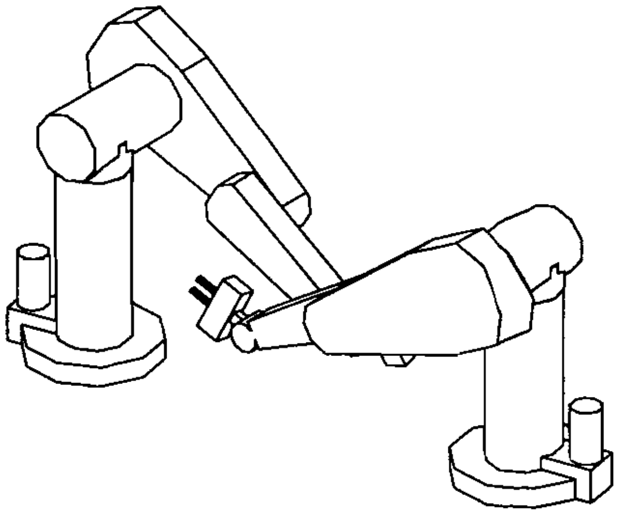


Figure 1: Two robots almost interfering

folklore of computational geometry; unlike the other two methods, it is also highly dependent on the types of geometric features allowed in the system. In the author's implementation these three methods operate in cascade, with each stage attempting to answer the NOD problem itself and only passing on parts of the original problem that it cannot tackle. Much of the efficiency of the overall implementation results from the control of this cascade.

The rest of this paper is organised as follows. The theory behind the S-bounds method is described in §2 in a fairly general way. §3 describes how this theory is used to prune the search space for the NOD problem, using the theory of redundancy [Til84]. §4 and §5 provide the details of the spatial subdivision and the exhaustive steps respectively, §6 gives some examples of interference detection within the ROBMOD geometric modelling system, and §7 provides a summary.

2 The Theory of S-bounds

2.1 Formalism

In a CSG modelling system objects are described either as primitive shapes, which are already known to the system, or as set-combinations[1] of other shapes together with rigid-body transformations. In general the modelling system representation is equivalent to a set of directed graphs, where: the arcs represent functional application; the non-terminal nodes represent set-combination or transformation operators; and the terminal nodes represent primitive shapes, which are specified by a set of parameters, or they refer to other shape graphs. A valid CSG graph may always be rewritten as an equivalent CSG tree in which the branches only represent set-combination operators and the leaves are transformed primitive shapes; this may be done by "sweeping" the transformation nodes towards the leaves, and copying references to other shapes as required. Although the algorithms described here can be modified to allow their use with the CSG graph structure itself, the theory is described in terms of CSG trees.

Thus, without loss of generality we can regard a CSG tree as a binary tree with branch nodes $\oplus$, $\otimes$ and $\ominus$, and leaf nodes which represent primitive shapes which we can label as convenient (e.g., P_n). We can write down these trees either as tree diagrams, or using an infix notation (e.g., $P_1 \oplus P_2$). Given a tree we may refer to a node in the tree as an entity in itself, or we may use the node to refer to the subtree that has that node as its root node—these nses can be distinguished by context.

As described, a CSG tree is a purely syntactic structnre, devoid of semantics. Most authors do not distinguish between such trees and their normal semantics, in which the trees are taken to mean the set formed by interpreting the leaves as the appropriate point sets, and the nodes as the appropriate set operations ($\oplus$ for $\cup$ or set union, $\otimes$ for $\cap$ or set intersection, $\ominus$ for $-$ or set difference). However such an approach is not appropriate here for three reasons. Firstly, the trees are rewritten as part of the algorithms to be described, and this rewriting is made clearer if it cannot be confused with manipulations of the semantics. Secondly, separating the semantics means that we can establish results that hold when the primitive objects are replaced by a number of different objects. And lastly, the S-bounds that form the core concept in the theory to be described in this section form an extra structure which is not easily appended to the old notation. (To help the reader we have summarised the extra notation used in figure 2.)

[1]There are several classes of set-operations; for the purpose of this paper we require that they form a Boolean algebra.

$\cup$, $\cap$, $-$, $\overline{}$	Set operations: union, intersection, difference, complement
$\oplus$, $\otimes$, $\ominus$, $\neg$	Tree node labels: union, intersection, difference, complement
$\mathcal{T}$, T	Root node or entire tree; tree node or subtree
ι, $\mathcal{I}$	Primitive and Full Interpretation Functions
β, β_Ω	Bounding Functions (β_Ω always returns Ω)
$\sqsubseteq$, $\sqsupseteq$	Partial orders on functions
$f[X \vdash S]$	Function equal to f, except that $f[X \vdash S](X) = S$
$\sqcup$, $\sqcap$	"Union" and "intersection" operations for S-bounds
$\perp$, $\top$	Tree node labels, which always interprete to $\emptyset$ and Ω resp.

Figure 2: Summary of Notation

Given a tree we define its semantics to be a subset of $\Re^n$, i.e., the 'shape of the tree'. To decide which subset, we need to know two functions: one is the *primitive interpretation* function, which takes a leaf node of the tree and returns the subset of $\Re^n$ that is the point set for the appropriate primitive; and the *bounding function*, which limits the attention of each node in the tree. We denote primitive interpretation functions by ι, ι', etc., and bounding functions, which take a node of the tree and return a subset of $\Re^n$, by β, β', etc. How the bounding functions are generated will be described in due course, but they can be thought of returning simple supersets of objects, such as boxes or spheres. (Snch bouuds are normally implemented as attributes attached to the CSG tree structnre itself, rather than as separate functions.) Given the primitive interpretatiou ι and bounding function β, we can compute the corresponding (full) interpretation function, $\mathcal{I}_\beta$, as follows:

- If T is a tree of the form $T_1 \left\{ \begin{matrix} \oplus \\ \otimes \\ \ominus \end{matrix} \right\} T_2$, then

$$\mathcal{I}_\beta(T) = \left(\mathcal{I}_\beta(T_1) \left\{ \begin{matrix} \cup \\ \cap \\ - \end{matrix} \right\} \mathcal{I}_\beta(T_2) \right) \cap \beta(T)$$

- Otherwise, T is a leaf node, and $\mathcal{I}_\beta(T) = \iota(T) \cap \beta(T)$

In our own work we have used the *regularised* set operations [Req80]; however all of the results in this section, and most of the results in the paper, only rely on the operations forming a Boolean algebra. Interpretations which are based on other primitive interpretations and bounding functions will be denoted accordingly; for example, $\mathcal{I}'_{\beta''}$ denotes the interpretation based on ι' and β''. The idea behind the bounding functions is that each bound (the value returned by the bounding function at each node) describes a subset of $\Re^n$ outside of which the value of the corresponding subtree should be ignored. Many geometric modelling systems form 'boxes' around the primitive nodes which are bounds in this sense; we have simply extended them to cover every node in the tree[2]. Note that for the trivial bounding function β_Ω, which always returns the universal set Ω, then the corresponding interpretation $\mathcal{I}_{\beta_\Omega}$ is always equivalent to the normal semantics of a CSG tree (i.e., one without the notion of a bounding function). For brevity, we write $\mathcal{I}(T)$ for $\mathcal{I}_{\beta_\Omega}(T)$.

In what follows we will introduce three classes of bounding functions, the last being S-bounds, which we will show to have the desirable properties of being easy to generate and which do not change the normal semantics of CSG trees. These particular bounds allow us to prune the subsets of space in which to search whilst performing NOD.

2.2 S-bounds

We distinguish three classes of bounding functions that do not change the semantics of a particular tree with respect to a particular primitive interpretation:

◇ Definitions

> Let T be a tree with primitive interpretation ι and bounding function β. We say that β is *consistent* (on T over ι) if $\mathcal{I}_\beta(T) = \mathcal{I}(T)$, and we say that β is *totally consistent* if $\mathcal{I}_{\beta'}(T) = \mathcal{I}(T)$, for all other bounding functions $\beta' \sqsupseteq \beta$. Here the notation $f \sqsubseteq g$, or $g \sqsupseteq f$, means that f and g are functions over the same domain with $f(T) \subseteq g(T)$ for all arguments T.

Consistent bounding functions preserve the meaning of a CSG tree. However, once we start using a consistent bounding function we must continue using it or risk changing the semantics of the tree. As a simple example, consider the "primitives" from $\Re^1$ given by $\iota(A) = [1,2]$, $\iota(B) = [0,3]$, and T the tree $A \ominus B$. Then a suitable consistent β is given by $\beta(T) = \Omega$, $\beta(A) = \beta(B) = \emptyset$, as then $\mathcal{I}_\beta(T) = \mathcal{I}(T) = \emptyset$. However, if we then selectively ignore the bound on A, effectively by setting its bound to Ω, then

[2]The University of Rochester's PADL2 modeller is another example that uses bounds at every node in the CSG tree.

the new bounding function, β', will no longer be consistent as $\mathcal{I}_{\beta'}(B) = \emptyset$, and $\mathcal{I}_{\beta'}(T) = \mathcal{I}_{\beta'}(A) = [1,2]$. Totally consistent bounding functions are the subclass of consistent bounding functions that do allow you to arbitrarily expand the bounds whilst staying consistent. (Apart from esoteric reasons for preferring totally consistent bounding functions there is a practical reason; when we apply rewrite rules to improve such functions we wish to be able to restrict ourselves to certain classes of elements, which is far easier if we can arbitrarily expand any bound.) β_Ω is a trivial example; a less trivial, and useful, class of totally consistent bounding functions is one that satisfies the following property:

◦ Definition

Given a tree T with primitive interpretation ι then a bounding function β satisfies the *boxing property* if

- $\beta(L) \supseteq \iota(L)$ for all leaf nodes L of T
- $\beta(T) = \Omega$ for all other nodes of T.

It is normally a simple matter to generate a set of simple bounds that satisfy the boxing property as the primitives in a CSG represention are themselves fairly simple. (For example, it is simple to find a box that contains a given cylinder.) Totally consistent bounding functions are useful, but are generally difficult to manipulate. The S-bound (for *Super-bounds*) bounding functions are a snbclass of the class of totally consistent bounding functions which are susceptible to mauipulation.

◦ S-bound Functions

Given a tree T with primitive interpretation ι the bounding function β is an S-bound function (on T over ι) if it is totally consistent with respect to all smaller primitive interpretations; i.e. if $\mathcal{I}'_{\beta'}(T) = \mathcal{I}'(T)$ for all $\beta' \sqsupseteq \beta$ and all $\iota' \sqsubseteq \iota$. The sets returned by an S-bonnd function are called S-bounds.

It is easy to see that β_Ω is always an S-bound bounding function, as is any bounding function satisfying the boxing property. Also, if β is an S-bound function then so is any $\beta' \sqsupseteq \beta$. We can now introdnce the two theorems that make all this preparation worthwhile; they describe rewrite rules for generating a new, better S-bound function from an old one. The notation $f[X \vdash S]$ denotes the function that is equal to f everywhere, except that the value at X is replaced by the value S.

◇ **The Upward Theorem**

Let $\mathcal{T}$ be a bounded tree with S-bound function β. If T is any subtree of $\mathcal{T}$ with immediate subtrees T_1 and T_2 then another S-bound function for $\mathcal{T}$ is given by β', where

$$\beta' = \beta[T \vdash S \cap \beta(T)],$$

and the set S is given by

$$S = \begin{cases} \beta(T_1) \cup \beta(T_2) & \text{if } T = T_1 \oplus T_2 \\ \beta(T_1) \cap \beta(T_2) & \text{if } T = T_1 \otimes T_2 \\ \beta(T_1) & \text{if } T = T_1 \ominus T_2 \end{cases}$$

◇ **The Downward Theorem**

Let $\mathcal{T}$ be a bounded tree with S-bound function β. If T is any subtree of $\mathcal{T}$, and T' is an immediate subtree of T, then another S-bound function for $\mathcal{T}$ is given by β', where

$$\beta' = \beta[T' \vdash \beta(T) \cap \beta(T')]$$

The proofs of these theorems are tedious, and they are deferred until the appendix. Given the intuitive idea of bounds being boxes that enclose the primitives in a CSG tree, the correctness of the Upward theorem should be evident; if we start with a bound set that satisfies the boxing property and just apply the Upward theorem, then the bounds that get formed at the branch nodes are just supersets of the sets that are formed when you take the normal semantics of the corresponding subtree's interpretation. Similarly, if we know that the set, represented by $\mathcal{T}$, can be enclosed within a box B, then it is "obvious" that we should be able to ignore any parts of the primitives of the tree that lie outside B. What is not as obvious, and is in fact quite difficult to prove, is that we can legitimately combine both types of rewrite rule within the same system. (The proofs are complicated by the existence of the $\ominus$ operator; they are almost trivial without it.) The advantage of the notational framework that has been built up in this section is that we can be quite precise about what rewrites are and are not allowed: this will be useful as we explain how S-bounds are useful within the framework of CSG.

2.3 A Two-Dimensional Example of S-bounds

To illustrate how S-bounds can be used consider the simple CSG tree shown in figure 3(a), consisting of a two operator tree and three primitives from $\Re^2$.

The primitives are each shown within a frame; this is simply to emphasis their relative layout. We have also shown the resultant shape described by the tree under the normal semantics.

The S-bounds we will use here will be rectangles aligned with the frame, and a set of S-bounds that obey the boxing property are shown in figure 3(b). If we apply the Upward Theorem twice, firstly about the $\oplus$ node and then about the $\otimes$ node, we get the new S-bounds shown in figure 3(c); as expected, each bound is a superset of the corresponding subtree. We can then "push" the bounds back down the tree by applying the Downward theorem four times, twice to the children of the $\otimes$ node, and then twice to the children of the $\oplus$ node; the resulting S-bonnds are shown in figure 3(d). The important feature here is that the S-bounds about the primitives are now smaller than the original bounds and, in particular, the S-bound abont the left-most primitive is now the null set. This reflects the fact that this primitive is redundant (in this configuration) [Til84], and could be removed from the CSG tree without affecting the resultant shape. (See §3 for more details.)

In fact it is possible to simplify the bonnds further by repeating the process of Upward and Downward Theorem applications, starting with the bounds in figure 3(d). This results in the other four bounds converging to the smallest non-null bound in figure 3(d). Although this is a simple example, the same principle applies to much larger CSG trees.

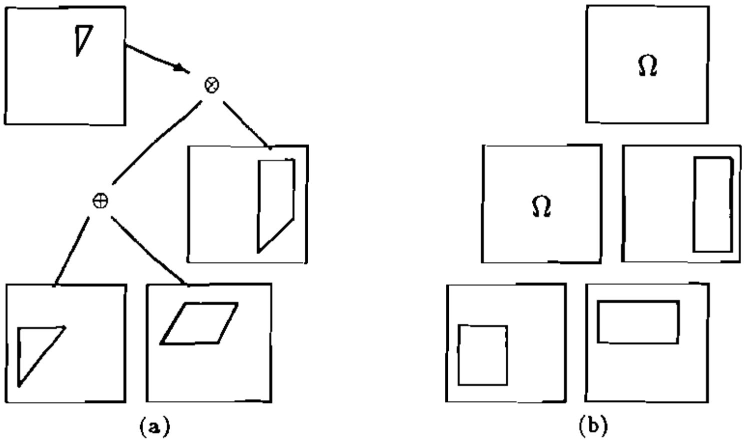
2.4 Making Use of S-bounds

So far we have only described S-bounds as an algebraic system; we have not described how they can be used in practise. The practical use that we have made of S-bounds is to use S-bound sets that are easily described, and manipulated quickly—in time linear in the size of the CSG description. This enables us to use S-bounds quickly as a preprocessing stage, to simplify the slower processing that follows. This philosophy applies to the NOD problem in particular, when large speedups in processing time are the norm, but it has also been applied by default to many of the other processing algorithms used in our geometric modelling system (e.g., drawing, inertial property calculation, and minimum distance calculations).

The S-bounds that are used are boxes, aligned with some arbitrary world coordinate axis system. Thus in two dimensions each box can be described as a four-tuple, viz.:

$$\langle x_l, y_l, x_h, y_h \rangle = \{(x,y) \mid x_l \leq x \leq x_h,\ y_l \leq y \leq y_h\}$$

We need a pair of operators, $\sqcup$ and $\sqcap$, such that $A \sqcup B \supseteq A \cup B$ and



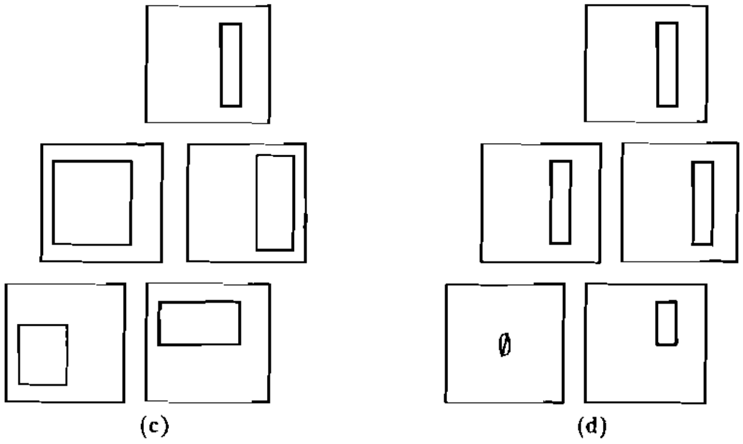


Figure 3: Two-Dimensional Example of S-bounds

$A \sqcap B \supseteq A \cap B$; we use the obvious pair:

$$\begin{aligned}\langle a,b,c,d\rangle \sqcup \langle \alpha,\beta,\gamma,\delta\rangle &= \langle \min(a,\alpha),\min(b,\beta),\max(c,\gamma),\max(d,\delta)\rangle\\ \langle a,b,c,d\rangle \sqcap \langle \alpha,\beta,\gamma,\delta\rangle &= \langle \max(a,\alpha),\max(b,\beta),\min(c,\gamma),\min(d,\delta)\rangle\end{aligned}$$

(We also need to denote infinite bounds, and to identify null bounds.) Such operators can obviously be applied in unit time; other properties of these operators include the identity $A \sqcap B = A \cap B$, and that both operators are commutative and associative, but neither distributes over the other.

Another obvious choice for the class of bounds we could use is spheres. We have not used them in our implementations, but we give a corresponding set of operators for completeness. A spherical bound is described by its centre and its radius, say $\langle \mathbf{c}, r\rangle$. Given two such spheres, say $\langle \mathbf{c}_l, r_l\rangle$ and $\langle \mathbf{c}_s, r_s\rangle$ with $r_l \geq r_s$, then if d is the Euclidean distance between $\mathbf{c}_l$ and $\mathbf{c}_s$:

- if the operation is $\sqcap$, we get three cases:
  - if $d > r_l + r_s$, the result is $\emptyset$;
  - if $d < r_l - r_s$, the result is the same as the smaller sphere;
  - otherwise the result is a new sphere, namely

$$\Big\langle \left[(d^2 + r_l^2 - r_s^2)\mathbf{c}_l + (d^2 - r_l^2 + r_s^2)\mathbf{c}_s\right]/2d^2,\\ \left[4r_l^2 r_s^2 - (d^2 - r_l^2 - r_s^2)^2\right]/4d^2\Big\rangle$$

- if the operation is $\sqcup$, we get two cases:
  - if $d \leq r_l - r_s$ then the result is the same as the larger sphere;
  - otherwise, the result is a new sphere, namely

$$\langle [(d + r_l - r_s)\mathbf{c}_l + (d - r_l + r_s)\mathbf{c}_s]/2d,\ (d + r_l + r_s)/2\rangle$$

In each case the centre of the resulting sphere lies on the line between the centres of the original spheres, and the sphere has minimum radius. These sphere operators are commntative, but not associative.

The next problem to solve is the order in which the rewrite rules (as given by the Upward and Downward theorems) are applied. We have used a simple ordering, namely we apply the Upward theorem in a bottom-up manner throughout the whole tree, followed by applications of the Downward theorem in a top-down manner. (This was the ordering used in the example of §2.3.) As was mentioned in §2.3 we can often gain by repeating this process; the bounds need not converge after the first applications. This order

```
procedure setSBs($n, T$);
    setboxes($T$);
    do $n$ times begin
        upSB($T$);
        downSB($T$);
    end
endproc

procedure downSB($T$);
    do-dSB($T, \Omega$);
endproc

procedure do-dSB($T, B$);
    $NB \leftarrow$ boundof($T$) $\sqcap B$;
    boundof($T$) $\leftarrow NB$;
    if not isa-leaf($T$) then begin
        do-dSB(leftchildof($T, NB$));
        do-dSB(rightchildof($T, NB$));
    end
endproc

procedure upSB($T$);
    if not isa-leaf($T$) then begin
        upSB(leftchildof($T$));
        upSB(rightchildof($T$));
        $L \leftarrow$ boundof(leftchildof($T$));
        $R \leftarrow$ boundof(rightchildof($T$));

        switch on operator-of($T$)
            case $\oplus$: $C \leftarrow L \sqcup R$;
            case $\otimes$: $C \leftarrow L \sqcap R$;
            case $\ominus$: $C \leftarrow L$;
        endsw
        boundof($T$) $\leftarrow$
                boundof($T$) $\sqcap C$;
    end
endproc

procedure setboxes($T$);
    if isa-leaf($T$) then
        attach a convenient bound
        that is a superset of $\iota(T)$
    else begin
        boundof($T$) $\leftarrow \Omega$;
        setboxes(leftchildof($T$));
        setboxes(rightchildof($T$));
    end
endproc
```

Figure 4: Code for S-bounds

of rewrite rule application is embodied in the procedure ***setSBs***(), which is sketched in fignre 4; this computes a set of S-bounds that satisfy the boxing conditions, and then repeatedly applies the Upward and Downward theorem throughout the tree a given number of times.

We have used aligned boxes in all of our work with S-bounds, although other types of bounds could be used. For example: boxes with arbitrary orientation; ellipsoids; convex hulls; and maintaining both a box and a sphere abont each node in a tree, and regarding their unevaluated intersection as an S-bound. An extreme case, which is of mainly theoretical interest, is to use the actual primitives themselves as initial S-bounds, i.e. $\beta(L) = \iota(L)$ for all leaf nodes L. However the gains to be had in obtaining "tighter" bounds have to be offset against the longer times required to compute them.

Given two S-bound functions, it is reasonable to ask whether they can be combined in some way. They can, by virtne of tbe following theorem, which is proved in the appendix:

◇ S-bound Intersection Theorem

> Let β_1 and β_2 be S-bound functions over some tree $\mathcal{T}$; then so is $\beta_1 \odot \beta_2$, where $(\beta_1 \odot \beta_2)(T) = \beta_1(T) \cap \beta_2(T)$ for all subtrees T of $\mathcal{T}$.

It is not necessary to use the algorithm shown in figure 4 to refine S-bounds; in practise this algoritbm works well in our applications, but other strategies could be tried. We can regard S-bounds as a constraint manipulation system in which information about individual constraints (i.e. the primitives) are passed to other subtrees. Applying heuristics to encourage the spread of tight constraints (i.e. small primitives and subtrees) could be fruitful.

2.5 Convergence Properties

In the general case a system of S-bounds need not converge at all, as there is no requirement that the rewrite rules be deterministic. (We can always choose an arbitrary superset of any S-bound.) For the S-bound sets and operators that we use, namely the aligned boxes, convergence in finite time is assured. To see this, note that each bound in $\Re^n$ can be described by $2n$ parameters, giving the maximum and minimum extent of the box in each dimension. Also, the operators $\sqcup$ and $\sqcap$ can only replace parameters for the bounds for one subtree by existing parameters from other bounds. Thus the total number of possible bounds is finite, and so the total number of possible S-bound functions that can be obtained by applications of the

Upward and Downward theorems is finite. Convergence now follows from the fact that the operations produce sequences of bounds which are (non-strictly) monotonically decreasing in size. In fact it is possible to show tbat for each Upward and Downward pass *either* this S-bound sequence converges, *or* we can set at least one S-bound value to $\emptyset$[3]. From this we can deduce that the aligned box system must converge in quadratic time [CY90]. Experimentally convergence of this system is normally quite rapid. We conjecture that, for practical purposes, a call of the form *setSBs*(d, T) will give a useful set of S-bounds, where d is the depth of T' and T' is obtained from T by compressing groups of $\oplus$ and $\otimes$ nodes into equivalent n-ary nodes—e.g., replacing a subtree of the form $A \oplus (B \oplus C)$ by a single tertiary node. For our own implementation we have set $d = 3$, as this seems to give good results and an S-bound processing algorithm that runs in time linear in the size of the CSG tree[4]. We can, however, construct artificial examples which require longer to converge fully. Consider the tree $\mathcal{T} = (I_0 \oplus I_2 \oplus \cdots \oplus I_{2n-2}) \otimes (I_1 \oplus I_3 \oplus \cdots \oplus I_{2n-1})$, where for simplicity we have not made the binary tree structure explicit and where I_m corresponds to the open interval $(m, m+1)$. Then sucessive calls to *upSB*$(\mathcal{T})$ followed by *downSB*$(\mathcal{T})$ result in the following sequence of bounds on the node $\mathcal{T}$: $(1, 2n-1)$, $(2, 2n-2)$, $(3, 2n-3)$,..., $(n-1, n+1)$, $\emptyset$. This convergence requires quadratic time, and so this example is a worst-case example for this problem.

2.6 Three-Dimensional Examples of S-bounds

We only present some simple examples of S-bound manipulations here, concentrating on the effect of $\ominus$ operators in CSG trees; more examples are to be found in later sectious. Figure 5(a) shows two loops, both created by taking a block and differencing out a cube to form the hole. The two loops can be mated (though not without cutting them), as shown in figure 5(b). Using S-bounds aligned with the blocks allows us to bound the space occupied by the intersection set, as shown in figure 5(c); figure 5(d) shows the total bound attained for a case in which the S-bounds are skewed with respect to the objects. (The thin lines show the outlines of the original objects.) As can be seen a useful reduction in the space to be considered is obtained; the reduction is not as dramatic as in some of our examples due partly to the limited reasoning possible through the $\ominus$ operators, and

[3] I am indebted to Chee Yap for this observation.

[4] When testing two robot assemblies we form the intersection of a pair of assemblies; each assembly is the union of a number of objects ("links") which are often themselves fairly simple objects.

partly to the true complexity of the problem. As another example, figure 6(a) shows a pair of objects, one of which is shaped like an 'E' on its side, and the other like a 'π', which are made up from taking the union of 4 blocks and 3 blocks, respectively. The two objects can be mated exactly, and we consider their intersection. With the bounds chosen to match the blocks all the primitives are shown to be null-bounded after 3 calls to *up-SB*() and *down-SB*(); figure 6(b) shows the remaining bounds after 2 calls, with the original bounds outlined as thin lines. Figure 6(c) shows the same objects, but this time made up by differencing out the "gaps" between the "teeth". In this case the S-bound set settles down to the set shown in figure 6(d); it is not possible to reason further about the interaction between the teeth.

2.7 Summary of S-bounds

We have demonstrated how a system of bounds can be established about every node in a CSG tree, and we have made explicit the semantics of these bounds. The bounds establish regions outside which the relevant subtrees do not matter *with respect to the entire tree.* S-bounds are used where we have a particular tree on which we wish to do computations, be they for drawing, mass properties, boundary evaluation, or whatever. For many applications the reduction in the size of the bounds is small (say of the order of a few percent), but even then they can lead to noticeable speedups in running times. However, when the operation to be performed is NOD we shall see that large increases in running speeds are the norm.

Boxing tests have become part of the folklore of computational geometry and solid modelling; indeed, they were the starting point for the research that lead to the approach adopted in this paper. However they have not before been placed on the firm theoretical footing shown here. The addition of our theoretical framework has two advantages: firstly, we have been able to show how to refine the bounds that are used (standard boxing tests being equivalent to just using a set of S-bounds that satisfy the boxing property); and secondly, we have established results that are independent of any particular domain. The domain that we have been interested in is the regularised set model of shapes and motions [Req80, Cam84], but the results of this section hold for any Boolean algebra. The only work of which we are aware that is of a similar flavour to our own was performed at the University of Rochester's Production Automation Project, namely the work on localisations [Til81], which inspired much of the work to be described in §3 and has culminated in the concept of active zones [RV89]. Active zones are related to S-bounds in that the active zone of a subtree is, effectively, *defined* to be the region outside of which the subtree "doesn't matter". Rossignac

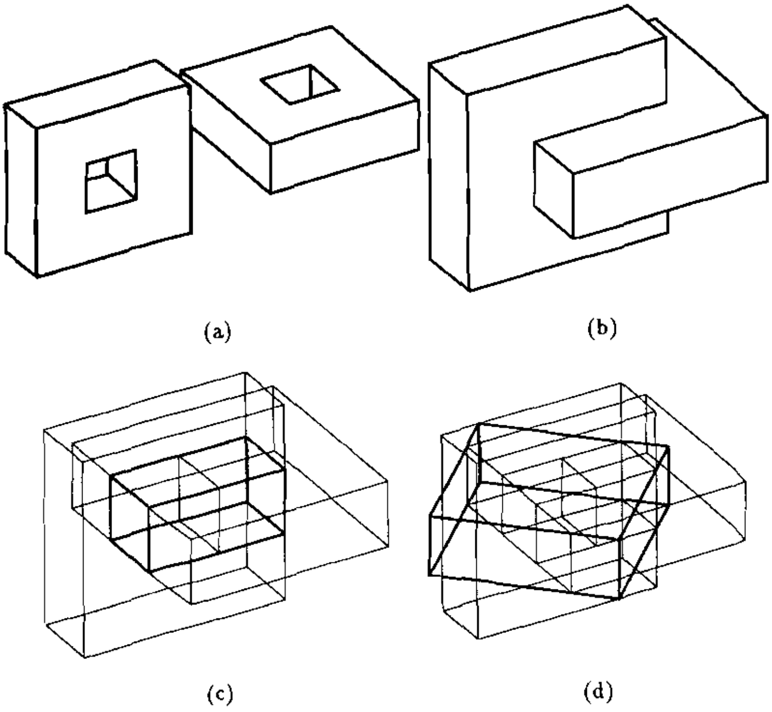


Figure 5: Two Intersecting Loops

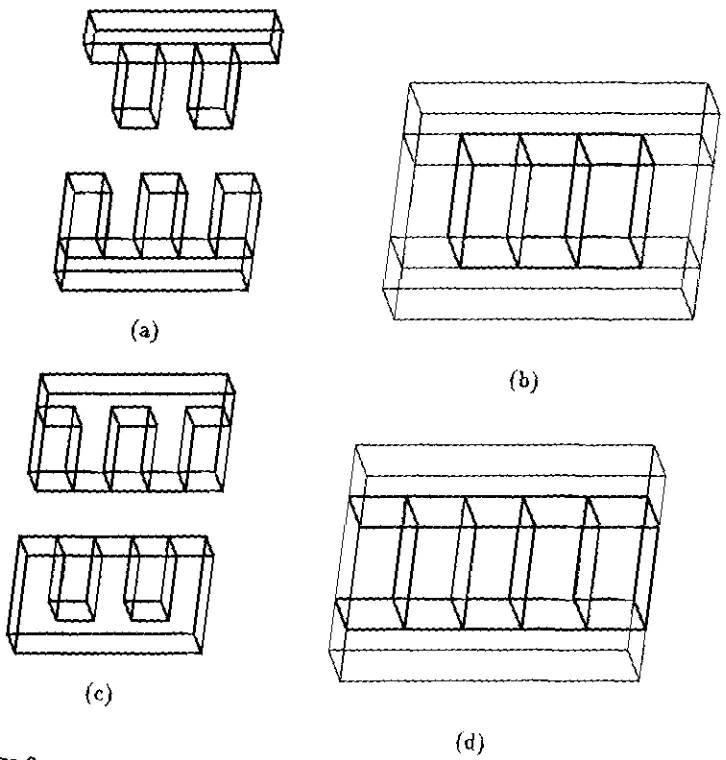


Figure 6: Effects of Using Different CSG Descriptions on S-Bounds

and Voelcker then show bow the active zone of a subtree can be computed, by use of an intermediate form (which is, effectively, a CSG tree with no internal $\ominus$ nodes). The active zone is, as defined, a single set, and so it does not in itself admit the types of fast processing using "approximations" to shapes that were the driving force behind the development of S-bounds. However active zones are a useful conceptnal tool, and are being used in the development of the PADL2 modeller.

Many B-rep modelling systems allow objects to be defined constructively; in such systems it would be possible to store the origiual construction information and then apply an S-bound analysis to tbat. This information conld then be used, say, for automatically scaling pictures of the object, or for marking the feature records in such a modeller with a box that can then be used when testing features for intersections. (By "feature" we primarily mean edge, face or surface patch records. Feature intersection tests are a commou component of many geometric algorithms, such as boundary evaluation.)

3 S-bounds and Null Object Detection

In this section we will build on the uotion of S-bounds to develop the top layer of our NOD algorithm. Giveu a CSG tree we run a S-bound processing algorithm to refine the bounds—in our own impleutation we have effectively been running $setSBs(3, \mathcal{T})$. Immediately we can note that any subtrees with a null S-bound (*null-bounded*) cannot affect the "value" of the tree, and so we can effectively prune any such subtrees from the CSG tree. In particular, if the whole tree is null-bounded *the whole tree must be null.* This is the sense in which the S-bound method is a partial decision procedure, and in onr experieuce with usiug the algorithm to perform interference detection for robotics it occurs suprisingly often. As an example, figure 1 shows two robot arms, and the corresponding CSG tree coutains 30 finite primitives and 8 infinite half-spaces; however, when the robots are checked for interference the S-bouuds preprocessing step alone is sufficient.

Having performed the S-bound step it is possible to jump straight to the division algorithm which is described in §4, using the S-bound attached to the root node of the tree as the spatial bound that that routine requires. However there is yet another stage that we can profitably iutersperse. This extra stage is based on Tilove's redundancy algorithm [Til84], although the version that we describe uses S-bounds directly. First, we need some extra notation. We assume that we are dealing with a particular tree, $\mathcal{T}$, so that any subtrees we discuss are subtrees of $\mathcal{T}$, and that a suitable S-bound refinement algorithm has been run on $\mathcal{T}$, such as *setSBs*().

◇ General Definitions

- Two subtrees are said to be *disjoint* if neither is a subtree of the other.
- The *order* of a subtree (in $\mathcal{T}$) is the number of times that the path from the root node of $\mathcal{T}$ to the root node of the subtree passes to the right of a $\ominus$ node. For example, in $\mathcal{T} = A \ominus (B \oplus (C \ominus D))$ A is of order 0, B and C are of order 1, and D is of order 2.
- A subtree is said to be *positive* if it is of even order, and *negative* otherwise.
- A subtree T is said to be *$\emptyset$-redundant*, or simply *redundant*, if $\mathcal{I}(\mathcal{T}) = \mathcal{I}_{\beta_\cap[T \vdash \emptyset]}(\mathcal{T})$; or equivalently, if the subtree T conld be replaced by a representation of the null set.

Tilove discovered in his seminal work on redundancy that all the positive primitives in a tree that represents the null set are $\emptyset$-redundant[5]. Further, if P is a positive primitive of $\mathcal{T}$ then P is $\emptyset$-rednndant if (in onr notation)

$$\mathcal{I}(\mathcal{T}) \cap \iota(P) = \emptyset. \tag{1}$$

The converse is not true; however, if the intersection is not null then neither is $\mathcal{I}(\mathcal{T})$, and so (1) can be used as the basis of a NOD algorithm:

◇ Tilove's Algorithm

> Pick a positive primitive of $\mathcal{T}$, and perform the test (1). If the test is false, then return false; otherwise replace the primitive by one whose interpretation is $\emptyset$, *simplify* $\mathcal{T}$, and repeat.

This algorithm is useful for two reasons. Firstly, the test in (1) is effectively the same as solving the NOD problem for $\mathcal{T}$ within the region given by $\iota(P)$; generally we can take advantage of this to ignore parts of $\mathcal{T}$ that lie outside $\iota(P)$, giving significant computational savings. (This is an example of a *spatial localization algorithm*, whereby we focus our attention on only part of the space that we are interested in.) Secondly, at each stage we either demonstrate that $\mathcal{I}(\mathcal{T}) \neq \emptyset$, or we can simplify the tree—that is, replace

[5]I am indebted to an anonymous reviewer for pointing out that [RV89] gives the basis of an extension to Tilove's ideas, which has been implemented into PADL2.

```
procedure Simplify(T);
  if boundof(T) = ∅ then
    return ⊥;
  if not isa-leaf(T) then
    L ← Simplify(leftchildof((T)));
    R ← Simplify(rightchildof((T)));
    rewrite T, if applicable
  return T
endproc
```

$$\begin{array}{lcl}
X \oplus \bot & \rightarrow & X \\
X \otimes \bot & \rightarrow & \bot \\
X \ominus \bot & \rightarrow & X \\
\bot \ominus X & \rightarrow & \bot \\
X \oplus \top & \rightarrow & \top \\
X \otimes \top & \rightarrow & X \\
X \ominus \top & \rightarrow & \bot \\
\top \ominus X & \rightarrow & \neg X
\end{array}$$

Figure 7: Simplifying CSG Trees

the tree by a smaller tree. Thus, if $\mathcal{I}(T) = \emptyset$ then we will go through all the positive primitives in turn, at each iteration the tree tested will get smaller, and we will end up with a tree that is *identically* null [Til84, Til81].

This is the first time we have discussed rewriting the CSG tree itself, and some explanatiou is required. So far we have simply rewritten the bounds on each node of the tree, using the rules for S-bouuds. From now on we will try to reduce the size of the CSG tree we are considering as we proceed. To do this, we add two new terminal nodes to those that can normally be found in a CSG tree, plns some new rewrite rules. The new terminal nodes are written $\bot$ and $\top$; $\bot$ represents a primitive whose interpretation is *always* $\emptyset$. and $\top$ a primitive whose interpretation is *always* Ω. With these nodes we can apply rewrite rules that reduce the size of the tree, based on identities such as $A \cup \emptyset = A$, etc. Figure 7 shows a partial set of such rules, together with a simple routine for applying them. For the purposes of this section only the rules concerning $\bot$ are of interest; $\bot$ can be introduced to replace any node whose bound is $\emptyset$.

3.1 Redundancy-Based NOD Algorithm

Our algorithm is based on Tilove's, but uses totally-consistent bounds as the focusing regions (generated from S-bounds), rather than primitives as the focusing regions. Proof of its correctness follows from the Redundancy Theorem, which is itself proven in the Appendix.

◇ Redundancy Theorem

> Let $\mathcal{T}$ be a tree with a totally consistent bounding function β. If T is any *positive* subtree of $\mathcal{T}$ such that $\mathcal{I}(\mathcal{T}) \cap \beta(T) = \emptyset$, then $\beta[T \vdash \emptyset]$ is another totally consistent bounding function on $\mathcal{T}$.

This theorem is our generalisation of (1), bnt the use of bounds is generally more convenient than using the shapes of the primitives themselves, and we are not limited to just considering leaf nodes of the tree. Note that the theorem does not hold if we replace both occurences of "totally consistent bounding function" with "S-bonnd function"; as a counter-example, consider $\mathcal{T} = A \ominus B$, with $\iota(A) = \iota(B) = X \neq \emptyset$. Theu $\mathcal{I}(\mathcal{T}) = \emptyset$ and β_Ω is an S-bound function on $\mathcal{T}$, but $\beta_\Omega[A \vdash \emptyset]$ is not an S-bound function—consider $\iota[B \vdash \emptyset]$.

Use of the Redundancy Theorem allows us to incrementally simplify the CSG tree as we cousider regions. We also need to be able to pick put nodes in the tree at which to apply the theorem; we choose the nodes from a *covering set.*

◇ Definitions

> Given a tree $\mathcal{T}$, a disjoint set of subtrees $\{T_1, T_2, \ldots, T_n\}$ is called a *covering set* (of $\mathcal{T}$) if
>
> $$\mathcal{T}{\downarrow}T_1{\downarrow}T_2{\downarrow}\ldots{\downarrow}T_n \equiv \perp$$
>
> where $\mathcal{T}{\downarrow}T_1$ means the tree $\mathcal{T}$ with the subtree T_1 overwritten by $\perp$, $\mathcal{T}{\downarrow}T_1{\downarrow}T_2$ means $(\mathcal{T}{\downarrow}T_1){\downarrow}T_2$, etc., and $\equiv$ is equivalence under the standard rewrite rules.

So if we use any covering set that includes only positive subtrees in our modified version of Tilove's algorithm then we are guaranteed to discover either that the tree does not represent the nnll set, or that

$$\mathcal{I}(\mathcal{T}) = \mathcal{I}(\mathcal{T}{\downarrow}T_1{\downarrow}T_2{\downarrow}\ldots{\downarrow}T_n) = \emptyset$$

and so the tree does represent the null set. Tilove used the set of all positive primitives as a covering set; in fact the set of zero-order primitives will also work, as will some smaller sets, discussed in §3.2. The general form of the redundancy algorithm is shown in *S-NOD*(), figure 8, in which *DC-NOD*() is the next layer of the NOD routine (discussed in §4).

```
procedure S-NOD($T$);
   $T \leftarrow$ Simplify($T$);
   $C \leftarrow$ coverset($T$);
   while $C \neq \emptyset$ and $T \not\equiv \perp$ do begin
      choose $T'$ from $C$;
      $C \leftarrow C - T'$;
      if not DC-NOD(boundof($T'$), $T$)
      then
         return false;
      $T \leftarrow T \downarrow T'$;
      $T \leftarrow$ Simplify($T$);
   end
   return true;
endproc
```

Figure 8: Redundancy-Based NOD Algorithm

3.2 Finding Covering Sets

A suitable covering set for $A \ominus B$ is $\{A\}$; for $A \otimes B$ we can use either $\{A\}$ or $\{B\}$; but for $A \oplus B$ we must use $\{A, B\}$ (or the root node). We can express these ideas as part of a non-deterministic procedure, *coverset*(), as shown in figure 9. *coverset*() can generate all the possible, minimal covering sets of a CSG tree; for *S-NOD*() we need to select just one. To ensure linear time complexity of *coverset*() we use some simple heuristics, based around a simple estimate of the efficiency of taking different choices. This estimate is simply the size of the S-bound attached to a node, whereby "size" we mean the diameter or volume of the bound—both work well. The reason for this is that a subtree with a large S-bound will probably intersect more primitives than one with a small S-bound, and be more difficult to prove redundant; thus, given a choice, we should choose the subtree with the smaller bound. Our rules to decide which choices to make in coverset() are:

- for a node of the form $A \otimes B$, choose the child node with the smallest bound;
- for a node of the form $A \ominus B$, choose A;
- for a node of the form $A \oplus B$, choose the child nodes A and B if $\kappa * size(A \oplus B) > size(A) + size(B)$, where *size*() is the size of the bound on the node. (The rationale here is to consider the subtrees A

```
procedure coverset($T$);
    if isa-leaf($T$) then
        return $T$;
    either begin
        return $T$;
    end
    or begin
        $L \leftarrow$ coverset(leftchildof($T$));
        $R \leftarrow$ coverset(rightchildof($T$));
        switch on operator-of($T$)
            case $\oplus$: return $L \cup R$;
            case $\otimes$: either return $L$
                            or return $R$;
            case $\ominus$: return $L$;
        endsw
    end
endproc
```

Figure 9: *coverset*()

and B if the bounds would get noticeably smaller, otherwise stop. We set $\kappa = 2$.)

We have experimented with other, more complicated heuristics, but with no significant increase in processing speed. As well as choosing which covering set to pick, we also impose an order on the covering set (i.e., turn it into a list) so that *S-NOD*() considers the nodes in order of increasing size; in this way, we consider nodes which are easiest to deal with first, leaving the harder nodes for when the tree has been rewritten to make it smaller. Examples of covering sets are shown in §6.

4 Spatial Subdivision

The routines described so far have produced a set of regions, together with a CSG tree for each region within which we need to solve the NOD problem. The classical way to proceed iu the three-dimensional case is to attempt to generate all the possible segments of any edges of the intersection set; if we suceed in generating any such segments then the set is not null. Unfortunately the number of possible edge segments is large, and so the naive version of this algorithm has a complexity of $O(n^4)$ [Til81]. However we can intersperse a stage of spatial subdivision to improve greatly the expected speed of the NOD routine. If we use *L-NOD*(R,T) to mean the generic NOD problem localised within a region R then

$$R = \bigcup_i R_i \qquad \Longrightarrow \qquad \textit{L-NOD}(R,T) = \bigwedge_i \textit{L-NOD}(R_i,T)$$

where $\bigwedge$ is the logical conjunction operator. That is, we can split the problem up spatially without affecting the answer. By itself this observation is of little use; however, if we can simultaneously reduce the size of the CSG trees being considered by each subproblem we can effect a reduction in computational complexity. A mechanism for reducing the size of these trees is that used in [WQ80, WQ84, WB86] for applying spatial subdivision to the problem of producing graphical representation of objects described constructively; much of this section is based on this work, but describes the modifications required to tackle NOD, and also presents an analysis of the usefulness of this stage.

4.1 Simplifying CSG Trees in Regions

As an example of this mechanism in action, consider the simple two-dimensional example in figure 10(a). In this a quadrilateral is described as the intersection of four *half-spaces*; that is, sets of the form $\{\mathbf{x} \mid f(\mathbf{x}) \leq 0\}$ where in this case each f is of the form $px + qy + d$. (We use linear half-spaces here for simplicity, but the technique described here will work with general half-spaces.) The quadrilateral is shown within the region of interest—in this case, a square. Figure 10(b) shows the boundaries of the individual half-spaces; the "matter" side of the half-spaces are labelled with their corresponding leaf nodes, and the entire quadrilateral corresponds to the binary tree $(A \otimes B) \otimes (C \otimes D)$. Within the original region this tree is the minimum that can be used to describe the quadrilateral. However, if we consider the quadrant labelled NE separately (figure 10(c)), it is clear that $A \otimes D$ is a sufficient *local* representation of the quadrilateral *within* this subregion. Similarly, $A \otimes B$, $B \otimes C$ and $C \otimes D$ are sufficient within the quadrants NW, SW, and SE respectively. We can automate the generation of these *localised* trees as follows. We start with a region and a CSG tree that is a valid representation of some object in the region. Given a subregion, then for each primitive in the tree we consider the corresponding half-space. If the half-space equation is always positive within the subregion then we can effectively replace the half-space with $\emptyset$; this we do by replacing the leaf node with $\perp$. Similarly, if the half-space equation is always negative within the region we replace the leaf node by $\top$. For the region NE in figure 10(c)

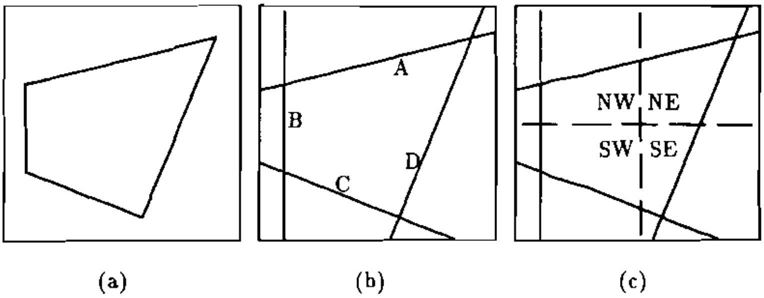


Figure 10: Two-dimensional example of tree localisation

this gives us the new tree $(A \otimes \top) \otimes (\top \otimes D)$. Now we can apply the rewrite rules shown in figure 7 to obtain the tree $A \otimes D$, as required.

Thus in general we can accomplish this localisation in two stages. At the first stage we identify which leaf nodes can be replaced by $\perp$ or $\top$. This is domain specific; however it is worth noting that if both the subregion and the shape represented by the leaf node are convex, then we can normally identify such nodes quickly. For example, if we have rectangular subregions and two-dimensional linear half-spaces, as in the example of figure 10, then the extreme values of the half-space function are achieved at the corners of the rectangle pointed at by the outward and inward pointing normals of the half-space boundary. For more complicated shapes we migbt not bother to check precisely whether the boundary of the shape intersects the region; for example, if we had a helical primitive we might enclose the helix in a cylinder for the purpose of testing. As long as our tests are conservative we will not simplify out any primitive that should not be removed from the tree. (It also ensures that this procedure is numerically well behaved.) If we have a bounded CSG tree then we can also take the bounds at each node into account; nodes whose bounds do not intersect the subregion can be replaced by $\perp$. This is the case with the NOD algorithm described, which produces totally consistent bonnds.

Tbe second stage, the tree rewriting, is purely synlactical. The rewrite rules shown in figure 7 are not in themselves complete, but it is not difficult to add the extra rules to accommodate the $\neg$ operator; for example, $\neg\top \rightarrow \perp$, $A \otimes \neg B \rightarrow A \ominus B$, etc. These rules can be applied top-down in the CSG tree to obtain a minimal tree in linear time. In our work we have not taken this course as we rewrite the trees to remove the $\ominus$ nodes before calling *DC-NOD*(); this is simply to reduce the number of cases that have to be considered and thus the size of the program. We also take this opportunity to resolve primitive objects (e.g., cuboids) into combinations of half-spaces (e.g., intersections of linear half-spaces). For convex primitives the bounds that were formed can normally be transferred to the half-spaces. The tree rewriting can be accomplished by: setting a sign flag at each node, indicating whether each subtree is positive or negative; replacing the operator $\ominus$ by $\otimes$ throughout the tree; swapping the operators at all negative binary nodes (i.e. $\oplus$ for $\otimes$ and vice-versa); and finally complementing the half-spaces at negative leaf nodes. The result of these tree rewriting steps is a tree with $\otimes$ and $\oplus$ binary nodes and half-spaces as leaf-nodes.

4.2 Performing the Spatial Subdivision

Given a set of subregions we know how to refine the trees; how do we decide on the subregions? Tilove [Til81] discusses a fixed set of regions, but Woodwark's group [WQ80] shows how to choose the regions dynamically. The latter approach has the advantage of being able to configure the set of regions so that more, small regions are used near parts of space that are complex. Experimentally this works well, and there is some theoretical justification for it—see section 4.3.

DC-NOD() is our routine for performing spatial subdivision for NOD, and it is outlined in figure 11. It is based on the popular "divide-and-conquer" paradigm; given a region and a tree it decides dynamically whether to "conquer" the problem or whether to do another spatial subdivision. Here *localise*() is a routine that performs the tree simplification steps outlined in §4.1, *can-do*() is the predicate that controls the subdivision, *base-NOD*() is the next layer of the NOD algorithm (discussed in §5), and *copytree*() *copies* the tree structure. The control predicate, *can-do*(), is the heart of the algorithm; it has to try to balance the cost of performing another subdivision step against the ease of being able to solve the NOD problem with the localised trees. For our implementations we have used simple regions and a simple method of subdividing the regions; in three-dimensions the regions are cuboids, and they are divided into eight octants. (This is not quite true, as the initial region is split into a number of roughly cubical regions in order to try to balance the problems.) [WQ84] describes a more complicated division strategy, whereby the choice of partition is influenced by the features of the objects described by the tree. Our version of *can-do*() checks to see whether the tree has one of a small number of very simple forms (see §5.1); then it just estimates the complexity of the tree by counting the number of half-spaces referenced. *can-do*() is actually a function of the size of the region being considered; for large regions, division is encouraged by only returning

```
procedure $DC\text{-}NOD(R,T)$;
   $localise(R,T)$;
   if $can\text{-}do(T)$ then
      return $base\text{-}NOD(R,T)$;
   else begin
      split $R$ into a partition $\{R_i\}$
      foreach $R_i$ do begin
         $T' \leftarrow copytree(T)$;
         if not $DC\text{-}NOD(R_i, T')$ then
            return false;
      end
   end
endproc
```

Figure 11: Spatial subdivision procedure for NOD

true for small trees, but as the region size gets smaller more complex trees can be passed on to *base-NOD*(). The idea here is that regions which are resistant to being simplified probably coincide with parts of the "object" that are truely complicated, and so we will probably not gain by trying to divide the problem further. This control strategy work well for our domain (i.e. robot workcells); in the general case we can adjust *can-do*() by trial and error to give a good performance.

There is some similarity between the action of this routine and the formation of a *quad tree* [Sam84]. The quad tree of an image is generated by looking at the complexity of the image within a square region, and then either storing a description of the image within that square, or splitting the square into four quadrants and describing those separately. One problem with quad trees is that it is difficult to perform a general rotation operation on them; this is not a problem with *DC-NOD*() as the division structure is not stored.

4.3 Computational Complexity of Spatial Subdivision

A worst-case analysis of the computational complexity of this spatial subdivision stage only shows that it will not dominate the complexity of the composite algorithm; experimentally this is extremely pessimistic. On the other hand calculating the expected complexity is difficult, partly because the analysis required is not trivial, but also because it is difficult to characterise the inputs to the algorithm, as we do not have a statistical model for a "typical" NOD problem. Thus we shall present only a simplistic, but nevertheless useful, analysis of the expected complexity.

Consider the case where the region of interest is a unit square and, furthermore, the only primitive shapes are squares, of varying sizes, which are aligned with the region of interest. Let the initial tree given as input to *DC-NOD*() contain N leaf nodes. Furthermore, let us consider only the simplest version of the control predicate *can-do*(), namely one that counts the number of leaf nodes in the initial tree and, on the basis of that count only, chooses a number D so that the initial unit square is divided into a $2^D \times 2^D$ grid by a uniform pattern of recursive calls. At each division stage a square is taken and divided into four quadrants, and so there will be $M = 4^D$ final regions. Each call to *DC-NOD*() involves an immediate time cost that is linear in the size of the tree that it is given, and results in either a call to *base-NOD*() or in four further calls to *DC-NOD*(). Thus the total time cost for the division process is proportional to:

> the sum, over all invocations of *DC-NOD*(), of the size of their inputs
>
> *which is proportional to*
>
> the sum, over all invocations of *DC-NOD*(), of the number of primitives in the simplified trees for each region
>
> *which is proportional to*
>
> the sum, over all primitives, of the number of regions corresponding to a call to *DC-NOD*() that intersect the boundary of the primitive

Thus in estimating the time cost we can scale up the cost of a single primitive.

A primitive is in a simplified tree if its boundary crosses the corresponding region. For large primitives (of similar size to the initial unit square) the number of regions affected from a uniform grid of size $2^d \times 2^d$ is approximately proportional to the length of the perimeter of the primitive divided by the diameter of the regions—i.e., 2^d. Thus the total cost for the large primitive is approximately $1+2+4+\cdots+2^D$, which is $O(2^D) = O(\sqrt{M})$. For small primitives (i.e. of size comparable with the final grid size, or smaller) each primitive will only affect a bounded number of regions at each division stage, with total cost of $O(D) = O(\log M)$. (This result has similarities with the result given in [Sam80] for generating a quad tree.) This gives a total time bound for all N primitives that is $O(N\sqrt{M})$; this compares favourably with simply choosing an initial grid and calling *base-NOD*() M times with the initial tree, which has a cost of $O(NM)$. As we are restricting ourselves to fixing M at the first call of the procedure we can estimate the costs of using different formulae for M; choosing M as $O(N)$, which we believe is reasonable, gives a time bound of $O(N^{3/2})$ for the division process; choosing M as $O(N^2)$, which we believe is pessimistic, gives a time bound of $O(N^2)$. Extending the analysis to 3 dimensions we see that large primitives will be taken into account in regions which straddle the boundary of the primitives, which gives a time bound for these primitives of $O(NM^{2/3})$; using the values for M suggested above gives a time bound of $O(N^{5/3})$ and $O(N^{7/3})$; in four dimensions the corresponding bounds are $O(NM^{3/4})$, $O(N^{7/4})$ and $O(N^{5/2})$.

We do not present here an analysis of time complexity for the procedure *base-NOD*(). [Til81] and [Cam84] argue that, under some reasonably general restrictions on the spatial distributions of the primitives in the CSG tree, if we choose $M \propto N$ in the analysis above then the complexity of algorithm will be asymptotically bounded by the cost of performing the spatial subdivision. We can illustrate this behaviour by noting that we could set up the control predicate, *can-do*(), to bound the size of CSG trees considered by *base-NOD*(), and so we can envisage this latter routine as always returning within some unit time. Of course, to do this we have to allow *DC-NOD*() to consider regions smaller that the ones suggested above; however in this case we would also expect larger terminal regions where the complexity of the space was simpler and these would tend to balance the total cost. Experimentally this optimism seems well jnstified.

5 Exhaustive Methods

The routines described so far have tried their best to avoid looking closely at the geometry described by the CSG tree. In our experience they generally succeed in pruning down the amount of $\Re^n$ that has to be examined in detail, as well as considerably reducing the size of the trees. This section concerns ways of implementing our routine *base-NOD*(), which takes as input a CSG tree and a region of $\Re^n$ in which to look. The routine can be thought as a theorem prover, which has been asked to prove a theorem of first-order predicate calculus. The theorem is of the form, "there exists no point $\mathbf{x}$ which is *inside* the object defined by the given CSG tree (within the region given)." The implementation of *base-NOD*() is highly domain specific, and the general techniques are described elsewhere (e.g., [Til84, Til80, Bro82]), but for completeness we present an overview of our implementation of *base-NOD*() which concentrates on the aspects which are amenable to use in other (geometric) domains. There are two paradigms which we have found useful—the *syntactic* paradigm and the *generate-and-test* paradigm.

5.1 Syntactic Paradigm

Some formulae of the predicate calculus are independent of the values taken on by their arguments; these are the tautologies and contradications of the propositional calculus. These occur rarely in general CSG trees, but are much more frequently found among the trees given to *base-NOD*() and the trees generated by the point classification routines (§5.2). The simplest, and most frequent, examples are the trees $\perp$ and $\top$. Trees consisting of a single half-space are also common. More complicated examples can only be detected if we identify equivalent and complementary half-spaces (or primitives) at the leaves of the CSG-trees. For example, if we have a tree of the form $A \otimes B$ and if we discover that A corresponds to the half-space $x \leq 0$ and B to $x \geq 0$ then we can established the nullity of the tree by reference to the contradiction $X \cap \overline{X}$ (in a regularised set system). In our own system we identify such leaf nodes as part of the preprocessing stage described in §4.1 by numerically sorting the half-spaces, which is an $O(n \log n)$ process. This numerical comparison is practical when we are dealing with simple half-spaces, such as linear half-spaces, but identifying such leaf nodes in general domains could be difficult, due to rounding errors in the computer arithmetic. In such cases we could exploit meta-knowledge about surfaces; for example, if we have a robot planning system that knows that a robot will establish a face/face *spatial relationship* when it places an object on a surface, there is no need to perform a geometric comparison to establish that the face equations are then related [AP75]. Once we have identified such leaf nodes we check for the existence of a tautology or contradiction only if the number of distinct half-spaces N is small (say ≤ 5) and then by explicitly evaluating all 2^N possible truth values. This is not as *ad hoc* as it may at first appear; the regions passed to *base-NOD*() generally contain only a small number of geometric features, each of which correspond to a small number of half-spaces. The only common case in which larger numbers of half-spaces are found is when features are mated, in which case spatial subdivision is unable to reduce the size of the CSG trees, but the number of distinct half-spaces is still small. Such cases are common in robotic assemblies. This procedure could also be used to transform the CSG tree into conjunctive or disjunctive normal form and so guide the search for test points in the generate-and-test paridigm.

5.2 Generate-and-Test Paradigm

The standard way of performing NOD in $\Re^3$ is to generate a set of possible edge-segments for the "object" and then to check to see if any really exist—if they do, the test returus **false**. This is an example of the generate-and-test paradigm.

One method of generating points for testing would be to choose them randomly. This has the advantage of simplicity, but also the distinct disadvantage of never terminating if the ohject *is* null. (However it could be used as a quick step if we had a case in which we expected the object *not* to be null.) To get around this non-termination we have to choose a finite and sufficient set of points. A non-null regular set must contain interior points, and as these are normally simpler to classify than boundary points it would be nice if we could choose interior points for our test set. Unfortunately it is difficnlt to generate such points *a priori*, and so points in the test set are likely to be on the boundary of the object. The "standard" technique in $\Re^3$ is to take pairs of surfaces that bound the primitives, and intersect these to form candidate *lines*; these are in turn intersected with surfaces to form candidate *edge segments*; and these edge segments are then classified (as being *inside*, *outside* or *surface*). This generates a sufficient set of points hecause a (bounded) non-null three-dimensional set must have a two-dimensional boundary, which in turn must be a collectiou of surface patches which are bounded by edge-segments from the generated set. However it should be realised that *any* sufficient set of points could be used, such as points which would be interior to a surface patch or points which are candidate vertices; there is a compromise between ease of geuerating the set and the difficulty in classifying the points within it.

To classify a given point we may proceed as follows. We have a CSG tree; if we imagine a small region arouud the given point then we will see that we can use a limiting form of the tree simplification routine (§4.1) to form a new CSG tree valid about the point. (Instead of testing half-spaces against the corners of an arbitrarily small box we just evaluate the half-space functions at the point and reject any whose absolute value is larger than some small ε.) Immediately we may discover that the equivalent CSG tree is $\perp$ or $\top$, corresponding to points that are *outside* or *inside* the ohject respectively. Otherwise we have to compute a local map of the region around the point—a so-called *neighbourhood* computation [Til80, Bro82] Intuitively, a neighbourhood is a map that is valid in some arbitarily small region about the point. As an example, consider figure 12(a), which shows a slotted block. If we want to classify the point shown, which lies on an edge, then a suitable neighbourhood map is shown in figure 12(b); it is two-dimensional,

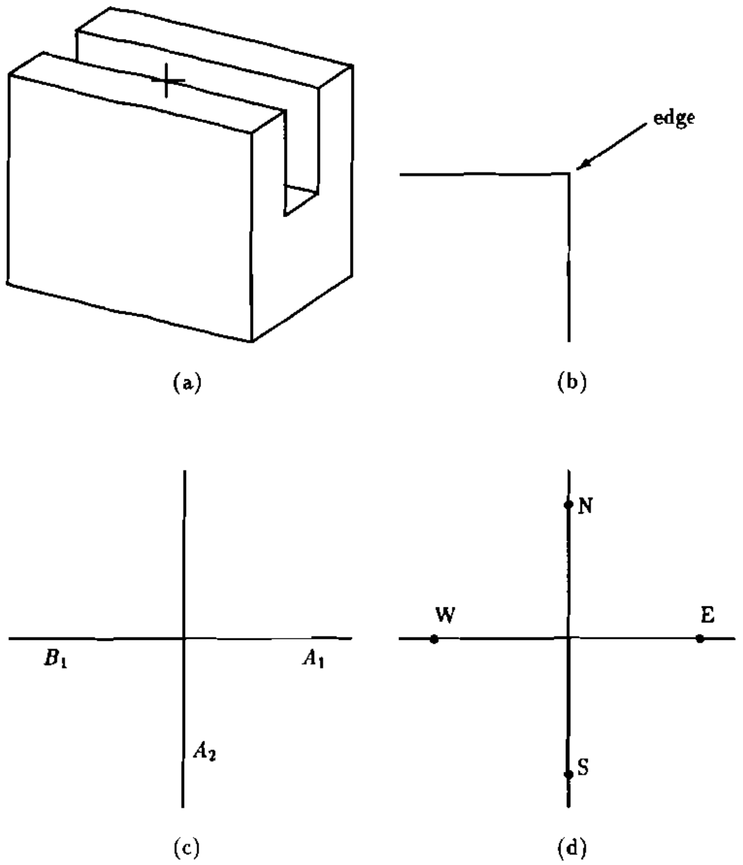


Figure 12: An Example of a Neighbourhood Computation

as the edge "looks the same" in the direction of the edge, and it is accurate within some arbitrarily small sphere, centred on the point. (Note that we cannot always rely on being able to make a neighbourhood map using linear entities; in particular two curved surfaces may have the same normal at the test point, and will then have to be sorted by curvature.) If the slotted block is described in the natural way as the difference of two blocks then the simplified CSG tree for the neighhourhood will be of the form $B_1 \ominus (A_1 \otimes A_2)$, where B_1 is the half-space of the larger block which forms the top of the slotted block, A_1 is the corresponding half-space for the smaller, differenced block, and A_2 is the half-space of the smaller block that forms the wall of the slot. These half-spaces are labelled in figure 12(c), with the labels on the "matter" side of the half-space boundaries; note that B_1 and A_1 are equivalent as half-spaces. To classify the point, imagine walking around the point in the neighbourhood map, and stopping each time we cross the boundary of a half-space. For our example, this might be at the points N, E, S and W shown in figure 12(d). If at each point at which we stop we take our existing local CSG tree, and simplify it again with respect to our new point, then the only possible results are $\top$, $\bot$, or some set-combination of half-spaces with the same boundary which is easily evaluated, possibly using the syntactic paradigm. Continuing with the example of figure 12, the point N is outside B_1 and A_1, and so we refine our local tree as follows:

$$B_1 \ominus (A_1 \otimes A_2) \rightarrow \bot \ominus (\bot \otimes A_2) \rightarrow \bot$$

The point E gives $B_1 \ominus (A_1 \otimes \top) \rightarrow B_1 \ominus A_1$, which requires use of the syntactic paradigm to show its equivalence to the null tree; the point S yields $\top \ominus (\top \otimes A_2) \rightarrow \neg A_2$; and the point W yields $B_1 \ominus (A_1 \otimes \bot) \rightarrow B_1$. The last two are not equivalent to $\bot$, which is not suprising as they correspond to the real faces bounding the edge. For the purpose of NOD we only have to look out for any evidence of non-nullity, which is easily discovered from the procedure above.

Thus it may be seen that the classification process is essentially one of reducing the dimensionality of the problem in order to be able to manipulate them within the discrete, linear memory of a computer. In the case of classifying the edge segments we chose a point interior to the segment so that we could reduce the problem to classifying a general point in two-dimensions, namely within the confines of an arbitrarily small "disc"—the neighbourhood map. In turn we could produce a local map for the disc, and then if necessary move away from the centre of the disc along the (projected) half-space boundaries to find points for which the classification problem is equivalent to solving for a point in a one-dimensional space; this last problem has a simple solution.

This necessity of performing dimensional reduction explains why we did not take the set of *candidate vertices of the object* (in $\Re^3$); classifying such points entails considering a neighbourhood that is topologically equivalent to a sphere. Similarly we see that it would be easier still to classify points that can only lie interior to surface patches, as then we only have to look at a line that pierces the patch to perform the classification. However there is, in general, no easy way to generate such points as we would first have to intersect all pairs of surface patches!

In the case of $\Re^4$ we use a similar analysis. First, to generate a set of candidate points we take all *triples* of half-space boundaries to form candidate lines, then use the half-space boundaries again to produce candidate edge-segments. Testing points within these segments is equivalent to testing a general point in three-dimensions, and so we can use the mechanisms described above to effect further reductions in problem dimensionality.

Finally, we may note that this exhaustive process can be viewed as exploring a search space. The first branches in our process tree correspond to the computation of the sufficient set of test points, and the later branches correspond to testing further points which are generated by the dimensional reduction mechanism. The only order for exploring this search space that we have tried is depth-first search; this is due mainly to the problems with memory management in the current incarnation of our system. Other search strategies could prove fruitful in situations where we expect the object not to be null.

6 Interference Detection in ROBMOD

The routines described in this paper have been implemented into a geometric modelling system called ROBMOD [CA88, Cam84]. ROBMOD is a CSG system which also has provisions for producing boundary information, and has been used as a test-bed for research into the use of spatial reasoning for robotics, such as the collision detection problem [Cam85, Cam84], and as a geometric processor for the RAPT robot language [ACC86] and for the Oxford Autonomous Guided Vehicle Project [Cam88].

S-bounds, based upon boxes aligned with the world coordinate system, are used by default in most of ROBMOD's algorithms. As described in §2 we effectively run *setSBs*$(3, T)$ to set up the S-bounds. If we consider the situation shown in figure 1, where we are testing to see whether the intersection of the two robots is null, then this reduces the initial set of 38 primitive bounds (figure 13(a)) into the bounds shown in figure 13(b) after one call to *upSB*() and *downSB*(), and then to a null-bounded CSG tree after another two pairs of calls. Figures 13(c) and 13(d) show covering sets for this tree, generated after the equivalent of *set-SBs*$(1, T)$ and *set-SBs*$(2, T)$ respectively.

Figure 14(a) shows a different situation, where we are testing for intereference between a single robot and its environment. The environment consists of the table on which the rohot sits, a block, and 8 rods, the latter being included to clutter the robot's environment. Figure 14(b) shows the initial set of bonnds around the primitives; it can be seen that there are a considerable number of interferences between the bounds. These interferences serve to reduce the effectiveness of the S-bounds stage, but despite this the resultant covering set, figure 14(c), is a considerable improvement over testing the entire workspace. Aligning the S-bounds with the robot arm improves the situation considerably (figure 14(d)): if we move the elbow of the robot so that there is no longer any interference between the robot and the block then this alignment reduces the computation time required to prove non-interference by about one-third.

As a final example, consider testing for interference between a "ray" and an object. We can simulate this in our system by using a long, thin block in place of the ray; such a ray is shown intersecting a robot in figure 15(a). (The system is happy considering aritrarily thin "rays"; we have shown a reasonably thick ray for clarity.) Figure 15(b) shows the ray together with the intial S-bonnds around the robot; after a single pass a new S-bound set is obtained about the primitives (figure 15(c)), which is only slightly improved by further passes (figure 15(d)). S-bound processing could be used in this way in a ray-casting system; an alternative approach, which may be faster if we want to pass a large number of parallel rays into the model (as is generally the case in picture generation) would be to generate the S-bounds foreach ray in parallel. If we consider the thick ray of figure 15 then we note that the S-bounds created are valid for any 'thin' ray within the thick ray. In particular we could split the thick ray into four quarter-rays, copying the S-bounds for eacb quarter-ray and then performing further refinements. For a large number of rays this would distribute the cost of generating localised bounded trees, in the same way that the dynamic divide-and-conquer strategy employed in *DC-NOD*() distributes the cost of generating localised trees for *base-NOD*().

7 Summary

We have described a complete intersection detection routine that is composed of several separable stages. The first stage consists of installing and then refining a set of bounds about the nodes in the CSG tree; these bounds

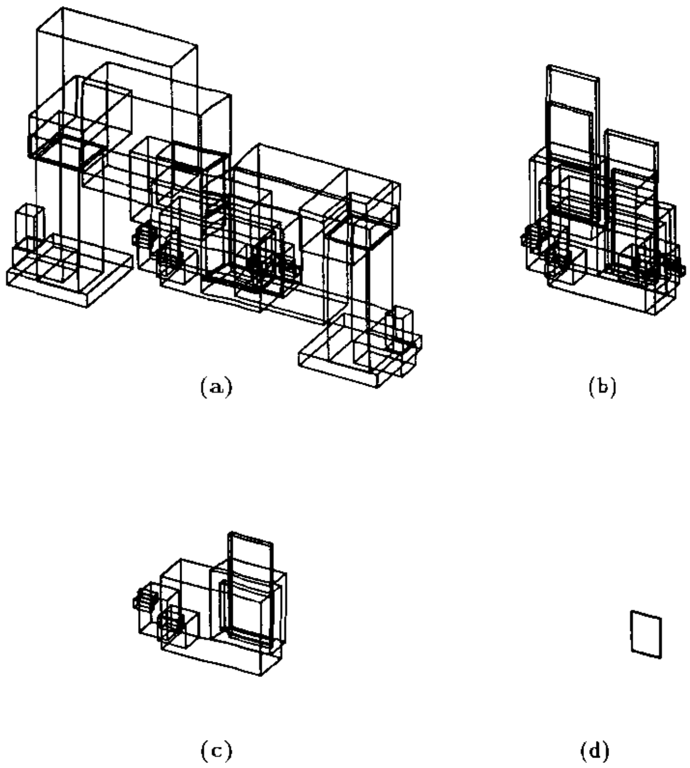


Figure 13: Various primitive bounds for the two robots

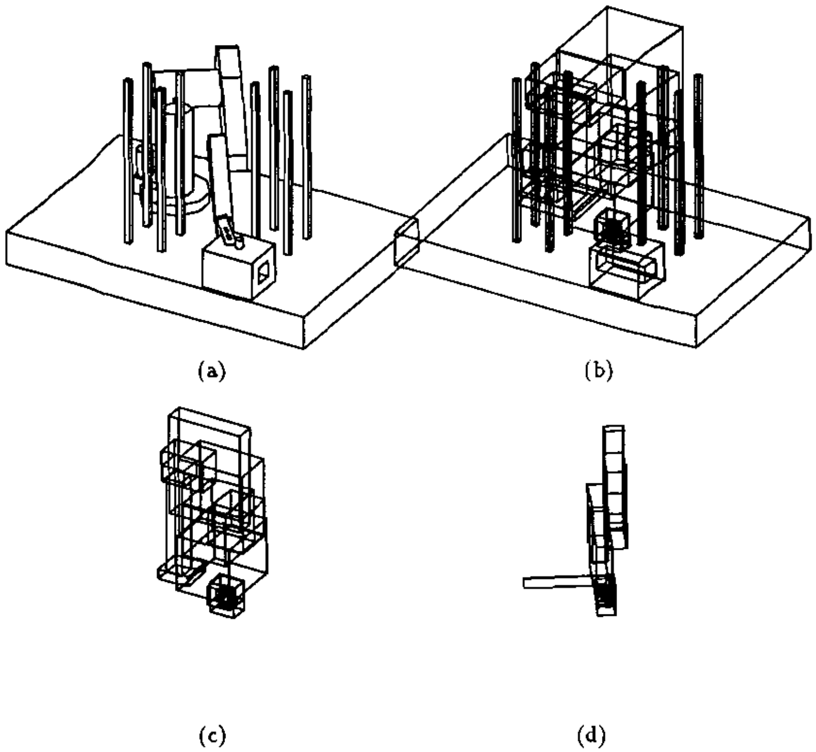


Figure 14: Robot tested in a cluttered environment

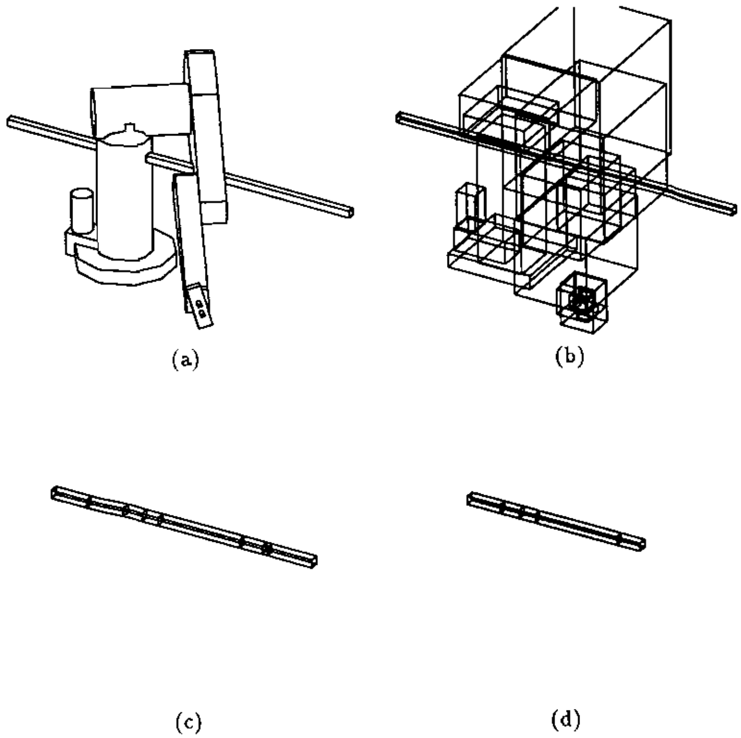


Figure 15: Ray Casting with S-Bounds

are based on the new theory of S-bounds. Once refined the bounds may prove sufficient to prove non-interference; otherwise the problem is split by considering a covering set for the tree, and considering the appropriate bound separately. This is a useful step because we are then able to incrementally simplify the CSG trees considered, using the concept of redundancy.

The next stage begins to consider the underlying geometry of the CSG primitives in more detail. We use a spatial subdivision method to split the problem spatially into a number of sub-problems. This technique is used to reduce the computational complexity of the routine significantly as it replaces one expensive problem by a number of (normally) much cheaper problems. Finally we may need to examine the geometry in detail. This part of the routine is the most domain-dependent. We have described a pair of paradigms that are reasonably general, namely one based on checking the form of Boolean functions, and one based on exploring the space looking for evidence of intersection. Other methods could easily be used in their place.

We have only detailed here algorithms that return a purely Boolean answer to the NOD or intersection question; in fact it is not too difficult to modify the routines to give some idea of the size and location of the intersection set. As described the S-bounds and spatial subdivision routines already find regions of space that might contain "matter", and in some cases where only a rough characterisation of the intersection set is required a list of those regions that turn out not to be empty may suffice. For a complete characterisation of the intersection set we would require domain-dependent (and tedious) modifications to the methods of §5 to return the appropriate boundary information.

We are wary about giving CPU timings for these routines because they are dependent on both the particular implementation and (more importantly) because they vary with the situations given. In many cases we have found that the S-bounds method is itself able to provide an answer *very* quickly. Otherwise, for our robotic workcells then as a rough rule-of-thumb we expect to see processing time increases of roughly a factor of five, two and thirty if we disable the S-bound refinement stage, the redundancy stage, and the spatial subdivision stage separately. However we should note that the stages do overlap in their effectiveness, as, say, the S-bounds refinement step tends to discard regions of space that would be easily processed by the spatial subdivision stage. Thus the total speedup for the entire routine, over just performing an exhaustive method with bounded primitives, is only a factor of about one hundred. On a SUN 3/160 workstation (without a floating-point accelerator) then we do expect the routine to return within one or two seconds in the worst cases seen.

Although we have described interference detection within a CSG modelling system it is clearly possible to carry out the S-bound and redundancy stages within a B-rep modeller provided that a set-combination tree is available for the objects. However the other two stages described are of more limited use in this case. We could perform spatial subdivision as described, but there is a considerable overhead in testing and refining the entities in the B-rep model.

Acknowledgements

Most of the work described in this paper was supported by an SERC studentship at the University of Edinburgh Department of Artificial Intelligence. The ideas were refined with the support of the McDonnell Douglas Independent Research and Development programme, and an SERC Atlas Research Fellowship. Many colleagues at Edinburgh, St. Louis and Oxford provided valuable feedback and suggestions. I am also indebted to Ari Requicha, Mike Brady and two anonymous referees, who made useful comments on drafts of this paper, to Jarek Rossignac for useful discussions regarding S-bounds and active zones, and to Chee Yap for his comments and his proof of the quadratic time convergence for the bounding box system.

References

[ACC86] A. P. Ambler, S. A. Cameron, and D. F. Corner. Augmenting the RAPT robot language. In U. Rembold and K. Hormann, editors, *Languages for Sensor-Based Control in Robotics*, pages 305–316, Castelvecchio Pascoli, September 1986. Springer-Verlag, ref. F29 (1987). Also as Univerity of Edinburgh DAI Research Paper 330.

[AP75] A. P. Ambler and R. J. Popplestone. Inferring the position of bodies from specified spatial relations. *Art. Intelligence J.*, 6(2):157–174, Summer 1975.

[Bro82] Chris Brown. PADL-2: a technical summary. *IEEE Comp. Graphics & Applications*, 2(2):69–84, March 1982.

[Bur69] R. M. Burstall. Proving properties of programs by structural induction. *Computer Journal*, 12(1):41–48, 1969.

[CA88] Stephen Cameron and Jon Aylett. ROBMOD: A geometry engine for robotics. In *IEEE Int. Conf. Robotics and Automation*, pages 880–885, Philadelphia, April 1988.

[Cam84] S. A. Cameron. *Modelling Solids in Motion*. PhD thesis, University of Edinburgh, 1984. Available from the Department of Artificial Intelligence.

[Cam85] S. A. Cameron. A study of the clash detection problem in robotics. In *IEEE Int. Conf. Robotics and Automation*, pages 488–493, St. Louis, March 1985.

[Cam88] S. A. Cameron. A geometric database for the oxford autonomous guided vehicle. In B. Ravani, editor, *CAD Based Programming for Sensory Robots*, pages 511-526, Castelvecchio Pascoli, July 1988. Springer-Verlag. Ref. F-50.

[CY90] S. A. Cameron and C. K. Yap. The use of bounds in geometric processing. Accepted for publication, ACM Transactions on Graphics, 1990.

[Req80] A. A. G. Requicha. Representations for rigid solids: Theory, methods and systems. *Computing Surveys*, 12(4), December 1980.

[RV82] A. A. G. Requicha and H. B. Voelcker. Solid modeling: A historical summary and contemporary assessment. *IEEE Comp. Graphics & Applications*, 2(2):9–24, March 1982.

[RV83] A. A. G. Requicha and H. B. Voelcker. Solid modeling: Current status and research directions. *IEEE Comp. Graphics & Applications*, 3(7):25–37, October 1983.

[RV89] J. R. Rossignac and H. B. Voelcker. Active zones in CSG for accelerating boundary evaluation, redundancy elimination, interference detection, and shading algorithms. *ACM Trans. Graphics*, 8(1):51–87, January 1989. Also as IBM Research Report RC13490, Yorktown Heights, NY, February 1988.

[Sam80] Hanan Samet. Region representation: Quadtrees from boundary codes. *Communications of the ACM*, 23(3):163–170, March 1980.

[Sam84] Hanan Samet. The quadtree and related hierachial data structures. *ACM Computing Surveys*, 16(2):187–260, June 1984.

[Til80] R. B. Tilove. Set membership classification: A unified approach to geometric intersection problems. *IEEE Transactions on Computers*, 29(10):874–883, October 1980.

[Til81] R. B. Tilove. *Expoiting Spatial and Structural Locality in Geometric Modelling*. PhD thesis, University of Rochester, October 1981. Available as TM-38, College of Engineering and Applied Science.

[Til84] R. B. Tilove. A null-object detection algorithm for constructive solid geometry. *Communications of the ACM*, 27(7):684–693, July 1984.

[WB86] J. R. Woodwark and A. Bowyer. Better and faster pictures from solid models. *Computer-Aided Engineering J.*, pages 17–24, February 1986.

[WQ80] J. R. Woodwark and K. M. Quinlan. The derivation of graphics from volume models by recursive division of the object space. In *Computer Graphics 80*, pages 335–343, London, August 1980.

[WQ84] J. R. Woodwark and K. M. Quinlan. Reducing the effect of complexity on volume model evaluation. *CAD J.*, 14(2), March 1984.

Appendix

For convenience in the proofs that follow we will define a class of auxiliary functions, ϕ_β, by the following rules:

- If L is a leaf node then $\phi_\beta(L) = \iota(L)$
- If T is a node of the form $A \left\{ \begin{array}{c} \oplus \\ \otimes \\ \ominus \end{array} \right\} B$ then

$$\phi_\beta(T) = \mathcal{I}_\beta(A) \left\{ \begin{array}{c} \cup \\ \cap \\ - \end{array} \right\} \mathcal{I}_\beta(B).$$

Then it is an easy proof (by structural induction [Bur69]) that $\mathcal{I}_\beta(T) = \phi_\beta(T) \cap \beta(T)$ for all trees T.

◇ The Upward Theorem

Let $\mathcal{T}$ be a bounded tree with S-bound function β. If T is any subtree of $\mathcal{T}$ with immediate subtrees T_1 and T_2 then another S-bound function for $\mathcal{T}$ is given by β', where

$$\beta' = \beta[T \vdash S \cap \beta(T)],$$

and the set S is given by

$$S = \begin{cases} \beta(T_1) \cup \beta(T_2) & \text{if } T = T_1 \oplus T_2 \\ \beta(T_1) \cap \beta(T_2) & \text{if } T = T_1 \otimes T_2 \\ \beta(T_1) & \text{if } T = T_1 \ominus T_2 \end{cases}$$

Proof It is sufficient to show that $\mathcal{I}'_{\beta''}(\mathcal{T}) = \mathcal{I}'(\mathcal{T})$ for any bounding function β'' with $\beta'' \sqsupseteq \beta'$, where β' is defined as above, and any interpretation $\iota' \sqsubseteq \iota$. The idea behind the proof is that if β'' is any such bounding function then we can find another bounding function $\beta^\dagger$ satisfying the conditions

(a) $\mathcal{I}'_{\beta''}(\mathcal{T}) = \mathcal{I}'_{\beta^\dagger}(\mathcal{T})$

(b) $\beta^\dagger$ is identically equal to β'' except that $\beta^\dagger(T) \supseteq \beta(T)$, $\beta^\dagger(T_1) \supseteq \beta(T_1)$, and $\beta^\dagger(T_2) \supseteq \beta(T_2)$.

Condition (b) implies that $\beta^\dagger$ is itself an S-bound function (as β is), and as $\beta^\dagger$ is equal to β'' except on T, T_1 and T_2 then it follows (from condition (a)) that $\mathcal{I}'_{\beta^\dagger}(\mathcal{T}) = \mathcal{I}'_{\beta''}(\mathcal{T})$, and so both are equal to $\mathcal{I}'(\mathcal{T})$.

To choose $\beta^\dagger$ we consider the three cases corresponding to the operator of T.

Operator is $\oplus$

Consider the sets $A = \beta''(T) \cup \beta(T)$ and $B = \beta''(T) \cup \beta(T_1) \cup \beta(T_2)$. Then $A \supseteq \beta(T)$, $B \supseteq \beta(T_1)$, $B \supseteq \beta(T_2)$, and

$$A \cap B = \beta''(T) \cup \{\beta(T) \cap [\beta(T_1) \cup \beta(T_2)]\} = \beta''(T) \cup \beta'(T) = \beta''(T)$$

as $\beta''(T) \supseteq \beta'(T)$. Thus

$$\begin{aligned}\mathcal{I}'_{\beta''}(T) &= \phi'_{\beta''}(T) \cap \beta''(T)\\ &= \Big\{[\phi'_{\beta''}(T_1) \cap \beta''(T_1)] \cup [\phi'_{\beta''}(T_2) \cap \beta''(T_2)]\Big\} \cap (A \cap B)\\ &= \Big\{[\phi'_{\beta''}(T_1) \cap \beta''(T_1) \cap B] \cup [\phi'_{\beta''}(T_2) \cap \beta''(T_2) \cap B]\Big\} \cap A\end{aligned}$$

So if we take $\beta^{\dagger}(T) = A$, $\beta^{\dagger}(T_1) = \beta''(T_1) \cap B$, and $\beta^{\dagger}(T_2) = \beta''(T_2) \cap B$ then $\mathcal{I}'_{\beta^{\dagger}}(T)$ will be identically equal to the expression above and $\beta^{\dagger}$ will be of the required form.

Operator is $\otimes$

Consider the sets $A = \beta''(T) \cup \beta(T)$, $B = \beta''(T) \cup \beta(T_1)$, and $C = \beta''(T) \cup \beta(T_2)$. Then $A \supseteq \beta(T)$, $B \supseteq \beta(T_1)$, $C \supseteq \beta(T_2)$, and

$$A \cap B \cap C = \beta''(T) \cup \{\beta(T) \cap \beta(T_1) \cap \beta(T_2)\} = \beta''(T) \cup \beta'(T) = \beta''(T)$$

Thus

$$\begin{aligned}\mathcal{I}'_{\beta''}(T) &= \Big\{[\phi'_{\beta''}(T_1) \cap \beta''(T_1)] \cap [\phi'_{\beta''}(T_2) \cap \beta''(T_2)]\Big\} \cap (A \cap B \cap C)\\ &= \Big\{[\phi'_{\beta''}(T_1) \cap \beta''(T_1) \cap B] \cap [\phi'_{\beta''}(T_2) \cap \beta''(T_2) \cap C]\Big\} \cap A\end{aligned}$$

and so choose $\beta^{\dagger}(T) = A$, $\beta^{\dagger}(T_1) = \beta''(T_1) \cap B$, and $\beta^{\dagger}(T_2) = \beta''(T_2) \cap C$.

Operator is $\ominus$

Consider the sets $A = \beta''(T) \cup \beta(T)$ and $B = \beta''(T) \cup \beta(T_1)$. Then $A \supseteq \beta(T)$, $B \supseteq \beta(T_1)$, and $A \cap B = \beta''(T)$. So

$$\begin{aligned}\mathcal{I}'_{\beta''}(T) &= \Big\{[\phi'_{\beta''}(T_1) \cap \beta''(T_1)] - [\phi'_{\beta''}(T_2) \cap \beta''(T_2)]\Big\} \cap (A \cap B)\\ &= \Big\{[\phi'_{\beta''}(T_1) \cap \beta''(T_1) \cap B] - [\phi'_{\beta''}(T_2) \cap \beta''(T_2)]\Big\} \cap A\end{aligned}$$

and so choose $\beta^{\dagger}(T) = A$, $\beta^{\dagger}(T_1) = \beta''(T_1) \cap B$ and $\beta^{\dagger}(T_2) = \beta''(T_2)$.

◇ The Downward Theorem

Let $\mathcal{T}$ be a bounded tree with S-bound function β. If T is any subtree of $\mathcal{T}$, and T' is an immediate subtree of T, then another S-bound function for $\mathcal{T}$ is given by β', where

$$\beta' = \beta[T' \vdash \beta(T) \cap \beta(T')]$$

Proof It is sufficient to prove that

$$\mathcal{I}'_{\beta''}(\mathcal{T}) = \mathcal{I}'(\mathcal{T})$$

for any bounding function $\beta'' \sqsupseteq \beta'$, and any interpretation $\iota' \sqsubseteq \iota$. Proceed by considering the cases of the set operation at T, but first, consider the sets $A = \beta''(T') \cup \beta(T)$ and $B = \beta''(T') \cup \beta(T')$. Then $A \supseteq \beta(T)$, $B \supseteq \beta(T')$, and

$$A \cap B = \beta''(T') \cup [\beta(T) \cap \beta(T')] = \beta''(T') \cup \beta'(T') = \beta''(T')$$

We also require the following two lemmas, which are presented here without proof. (Proof is by structural induction; they are proved in [Cam84].)

◇ Lemma Down1

Let T be a bounded tree with bounding function β and interpretation of primitives ι. If S is an arbitrary set and ι' is the interpretation given by

$$\iota'(P) = \begin{cases} \iota(P) \cap S & \text{if P is a zero-order primitive of } T \\ \iota(P) & \text{otherwise} \end{cases}$$

then $\mathcal{I}'_{\beta}(T) = \mathcal{I}_{\beta}(T) \cap S$.

◇ Lemma Down2

Let $\mathcal{T}$ be a bounded tree with two bounding functions $\beta^{\dagger}$ and $\beta^{\ddagger}$, and two interpretatious, $\iota^{\dagger}$ and $\iota^{\ddagger}$. If we are given a subtree T of $\mathcal{T}$, and a subtree T' of T such that

- $\beta^{\dagger}$ and $\beta^{\ddagger}$ are identical on all subtrees of T which are not subtrees of T',
- $\iota^{\dagger}$ and $\iota^{\ddagger}$ are identical on all primitives of T which are not primitives of T', and

- $\mathcal{I}^{\dagger}_{\beta\dagger}(T') \subseteq \mathcal{I}^{\ddagger}_{\beta\ddagger}(T')$,

then if T' is $\left\{ \begin{array}{c} \text{positive} \\ \text{negative} \end{array} \right\}$ in T, $\mathcal{I}^{\dagger}_{\beta\dagger}(T) \left\{ \begin{array}{c} \subseteq \\ \supseteq \end{array} \right\} \mathcal{I}^{\ddagger}_{\beta\ddagger}(T)$.

This lemma may be paraphrased by "making a positive subtree larger results in a larger tree, and making a negative subtree larger results in a smaller tree".

Operator is $\otimes$

Assume without loss of generality that $T = T' \otimes T''$. Then

$$\begin{aligned} \mathcal{I}'_{\beta''}(T) &= \left\{[\phi'_{\beta''}(T') \cap A \cap B] \cap \mathcal{I}'_{\beta''}(T'')\right\} \cap \beta''(T) \\ &= \left\{[\phi'_{\beta''}(T') \cap B] \cap \mathcal{I}'_{\beta''}(T'')\right\} \cap [\beta''(T) \cap A]. \end{aligned}$$

So consider the bounding function $\beta^{\dagger}$, which is identical to β'' except that $\beta^{\dagger}(T) = \beta''(T) \cap A$ and $\beta^{\dagger}(T') = B$. Then $\beta^{\dagger} \sqsupseteq \beta$ (and so is an S-bound function), $\mathcal{I}'_{\beta''}(T) = \mathcal{I}'_{\beta\dagger}(T)$, and $\beta^{\dagger}$ is identical to β'' outside T. So

$$\mathcal{I}'_{\beta''}(T) = \mathcal{I}'_{\beta\dagger}(T) = \mathcal{I}'(T)$$

as required.

Operator is $\oplus$

Assume without loss of generality that $T = T' \oplus T''$, and note that for *any* bounding function β

$$\mathcal{I}'_{\beta}(T) = \left\{\phi'_{\beta}(T') \cap \beta(T') \cap \beta(T)\right\} \cup \left\{\mathcal{I}'_{\beta}(T'') \cap \beta(T)\right\} \qquad (2)$$

Then consider the bounding functions $\beta^{\dagger} = \beta''[T' \vdash B]$ and $\beta^{\ddagger} = \beta^{\dagger}[T \vdash A \cap \beta''(T)]$. Then $\beta^{\dagger} \sqsupseteq \beta$ and $\beta^{\ddagger} \sqsupseteq \beta$, and so both are S-bound functions. Using identity (2), and noting that $\phi'_{\beta''}(T') = \phi'_{\beta\ddagger}(T') = \phi'_{\beta\dagger}(T')$, we have

$$\begin{aligned} \mathcal{I}'_{\beta\ddagger}(T) &= \left\{\phi'_{\beta''}(T') \cap A \cap B \cap \beta''(T)\right\} \cup \left\{\mathcal{I}'_{\beta''}(T'') \cap \beta''(T) \cap A\right\} \\ \mathcal{I}'_{\beta''}(T) &= \left\{\phi'_{\beta''}(T') \cap A \cap B \cap \beta''(T)\right\} \cup \left\{\mathcal{I}'_{\beta''}(T'') \cap \beta''(T)\right\} \\ \mathcal{I}'_{\beta\dagger}(T) &= \left\{\phi'_{\beta''}(T') \qquad \cap B \cap \beta''(T)\right\} \cup \left\{\mathcal{I}'_{\beta''}(T'') \cap \beta''(T)\right\} \end{aligned}$$

and so

$$\mathcal{I}'_{\beta\ddagger}(T) \subseteq \mathcal{I}'_{\beta''}(T) \subseteq \mathcal{I}'_{\beta\dagger}(T)$$

Using Lemma Down2, and defining $\Box$ to be $\subseteq$ if T is positive in $\mathcal{T}$, $\supseteq$ otherwise, we have

$$\mathcal{I}'_{\beta\dagger}(\mathcal{T}) \Box \mathcal{I}'_{\beta''}(\mathcal{T}) \Box \mathcal{I}'_{\beta\ddagger}(\mathcal{T}) \tag{3}$$

But $\beta^\dagger$ and $\beta^\ddagger$ are both S-bound functions, and so the outermost terms in (3) are both equal to $\mathcal{I}'(\mathcal{T})$, and

$$\mathcal{I}'_{\beta''}(\mathcal{T}) = \mathcal{I}'(\mathcal{T})$$

as required.

Operator is $\ominus$

If T can be written in the form $T' \ominus T''$ then the result holds as in the $\oslash$ case, as $\phi'_{\beta''}(T) = \mathcal{I}'_{\beta''}(T') \cap \overline{\mathcal{I}'_{\beta''}(T'')}$. So consider $T = T'' \ominus T'$. This case is by far the most difficult case to prove. (It is also the only case that requires S-bounds to be different from totally consistent bounding functions). Firstly, define the extra bounding functions $\beta^\dagger = \beta''[T \vdash \beta(T)]$ and $\beta^\ddagger = \beta''[T' \vdash \beta''(T') \cup \beta(T')]$. Then note that $\beta^\ddagger \sqsupseteq \beta$, and is thus an S-bound function. Secondly, define the interpretation

$$\iota''(P) = \begin{cases} \iota'(P) \cap \beta(T) & \text{if P is a primitive of zero-order in } T' \\ \iota'(P) & \text{otherwise} \end{cases}$$

and note that by Lemma Down1

$$\mathcal{I}''(T') = \mathcal{I}'(T') \cap \beta(T) \tag{4}$$

The main stages of the proof follow.

Step 1: Show that $\beta^\dagger$ is consistent

Consider $\beta^\dagger$, and note that for any interpretation $\iota^\circ \sqsubseteq \iota$

$$\begin{aligned}
\mathcal{I}^\circ_{\beta\dagger}(T) &= \left\{\mathcal{I}^\circ_{\beta\dagger}(T'') - [\phi^\circ_{\beta\dagger}(T') \cap \beta^\dagger(T')]\right\} \cap \beta^\dagger(T) \\
&= \left\{\mathcal{I}^\circ_{\beta''}(T'') - [\phi^\circ_{\beta''}(T') \cap \beta''(T')]\right\} \cap \beta(T) \\
&\qquad \text{by the definition of } \beta^\dagger \\
&= [\mathcal{I}^\circ_{\beta''}(T'') \cap \beta(T)] - [\phi^\circ_{\beta''}(T') \cap A \cap B] \\
&= \left\{[\mathcal{I}^\circ_{\beta''}(T'') \cap \beta(T)] - [\phi^\circ_{\beta''}(T') \cap B]\right\} \cup \left\{[\mathcal{I}^\circ_{\beta''}(T'') \cap \beta(T)] - A\right\} \\
&\qquad \text{by the identity } X - (Y \cap Z) = (X - Y) \cup (X - Z) \\
&= [\mathcal{I}^\circ_{\beta''}(T'') \cap \beta(T)] - [\phi^\circ_{\beta''}(T') \cap B] \\
&\qquad \text{as } \mathcal{I}^\circ_{\beta''}(T'') \cap \beta(T) \subseteq \beta(T) \subseteq A \\
&= \mathcal{I}^\circ_{\beta\dagger[T' \vdash B]}(T)
\end{aligned}$$

But $\beta^\dagger[T' \vdash B] \sqsupseteq \beta$, and so is an S-bound function, and $\beta^\dagger[T' \vdash B]$ is identical to $\beta^\dagger$ outside T, and so

$$\begin{array}{lrcll} & \mathcal{I}^\circ_{\beta^\dagger}(T) & = & \mathcal{I}^\circ_{\beta^\dagger[T' \vdash B]}(T) = \mathcal{I}^\circ(T) & \\ \text{i.e.} & \mathcal{I}^\circ_{\beta^\dagger}(T) & = & \mathcal{I}^\circ(T) \qquad \text{for any } \iota^\circ \sqsubseteq \iota. & (5) \end{array}$$

Step 2: Show $\mathcal{I}''(T) = \mathcal{I}'(T)$

Note that

$$\begin{array}{rcl} \mathcal{I}'_{\beta^\dagger}(T) & = & \left\{\mathcal{I}'_{\beta^\dagger}(T'') - \mathcal{I}'_{\beta^\dagger}(T')\right\} \cap \beta(T) \\ & = & \left\{\mathcal{I}'_{\beta^\dagger}(T'') - [\mathcal{I}'_{\beta^\dagger}(T') \cap \beta(T)]\right\} \cap \beta(T) \\ & & \quad \text{by the identity } (X - Y) \cap Z = (X - (Y \cap Z)) \cap Z \\ & = & \left\{\mathcal{I}'_{\beta^\dagger}(T'') - \mathcal{I}''_{\beta^\dagger}(T')\right\} \cap \beta(T) \qquad \text{by (4)} \\ & = & \mathcal{I}''_{\beta^\dagger}(T) \qquad \text{as } \iota''(P) = \iota'(P) \text{ for primitives of } T'' \end{array}$$

But $\iota''(P) = \iota'(P)$ for all primitives which are not primitives of T, and so

$$\mathcal{I}'_{\beta^\dagger}(T) = \mathcal{I}''_{\beta^\dagger}(T)$$

But $\beta^\dagger$ is consistent over ι' and ι'' (by (5)), and so

$$\mathcal{I}''(T) = \mathcal{I}'(T) \qquad (6)$$

Step 3: Show $\mathcal{I}''_{\beta''}(T) = \mathcal{I}'(T)$

Now consider $\beta^\ddagger$. $\beta^\ddagger$ is an S-bound function, and so

$$\begin{array}{lll} & \mathcal{I}''_{\beta^\ddagger}(T) = \mathcal{I}''(T) = \mathcal{I}'(T) \qquad \text{by (6)} & \\ \text{i.e.} & \mathcal{I}''_{\beta^\ddagger}(T) = \mathcal{I}'(T) & (7) \end{array}$$

Further

$$\begin{array}{rcl} \mathcal{I}''_{\beta^\ddagger}(T') & = & \mathcal{I}'_{\beta^\ddagger}(T') \cap \beta(T) \qquad \text{by Lemma Down1} \\ & = & \phi'_{\beta''}(T') \cap [\beta''(T') \cup \beta(T')] \cap \beta(T) \\ & & \quad \text{by the definition of } \beta^\ddagger \\ & = & \phi'_{\beta''}(T') \cap \{[\beta''(T') \cap \beta(T)] \cup [\beta(T') \cap \beta(T)]\} \\ & & \quad \text{by distributivity} \\ & = & \phi'_{\beta''}(T') \cap \{[\beta''(T') \cap \beta(T)] \cup \beta'(T')\} \\ & = & \phi'_{\beta''}(T') \cap \{[\beta''(T') \cup \beta'(T')] \cap [\beta(T) \cup \beta'(T')]\} \end{array}$$

$$
\begin{aligned}
&= \phi'_{\beta''}(T') \cap \{\beta''(T') \cap \beta(T)\} \\
&= \left\{\phi'_{\beta''}(T') \cap \beta''(T')\right\} \cap \beta(T) \\
&= \mathcal{I}'_{\beta''}(T') \cap \beta(T) \\
&= \mathcal{I}''_{\beta''}(T') \qquad \text{by Lemma Down1} \\
\text{i.e.} \qquad \mathcal{I}''_{\beta\ddagger}(T') &= \mathcal{I}''_{\beta''}(T') \qquad (8)
\end{aligned}
$$

But $\beta^{\ddagger}$ and β'' are identical outside T', and so (8) implies that

$$\mathcal{I}''_{\beta\ddagger}(\mathcal{T}) = \mathcal{I}''_{\beta''}(\mathcal{T})$$

and then (7) implies

$$\mathcal{I}''_{\beta''}(\mathcal{T}) = \mathcal{I}'(\mathcal{T}) \qquad (9)$$

Step 4: Bracket $\mathcal{I}'_{\beta''}(\mathcal{T})$

$$\mathcal{I}'_{\beta\dagger}(T) = \phi'_{\beta\dagger}(T) \cap \beta^{\dagger}(T) = \phi'_{\beta''}(T) \cap \beta''(T) \cap \beta(T) = \mathcal{I}'_{\beta''}(T) \cap \beta(T)$$

and so

$$\mathcal{I}'_{\beta\dagger}(T) \subseteq \mathcal{I}'_{\beta''}(T) \qquad (10)$$

Also, by Lemma Down1, $\mathcal{I}''_{\beta''}(T') = \mathcal{I}'_{\beta''}(T') \cap \beta(T)$, and so

$$
\begin{aligned}
\mathcal{I}''_{\beta''}(T') &\subseteq \mathcal{I}'_{\beta''}(T') \qquad \text{which implies} \\
\mathcal{I}'_{\beta''}(T) &\subseteq \mathcal{I}''_{\beta''}(T) \qquad (11)
\end{aligned}
$$

Combining (10) and (11) gives

$$\mathcal{I}'_{\beta\dagger}(T) \subseteq \mathcal{I}'_{\beta''}(T) \subseteq \mathcal{I}''_{\beta''}(T) \qquad (12)$$

Bnt by Lemma Down2, (12) implies

$$\mathcal{I}'_{\beta\dagger}(\mathcal{T}) \Box \mathcal{I}'_{\beta''}(\mathcal{T}) \Box \mathcal{I}''_{\beta''}(\mathcal{T})$$

where $\Box$ is $\subseteq$ if T is positive in $\mathcal{T}$, and $\supseteq$ otherwise. But by (5) and (9) both of the outermost terms in this relationship are equal to $\mathcal{I}'(\mathcal{T})$, and so $\mathcal{I}'_{\beta''}(\mathcal{T}) = \mathcal{I}'(\mathcal{T})$, as required.

◇ S-bound Intersection Theorem

Let β_1 and β_2 be S-bound functions over some tree $\mathcal{T}$; then so is $\beta_1 \odot \beta_2$, where $(\beta_1 \odot \beta_2)(T) = \beta_1(T) \cap \beta_2(T)$ for all subtrees T of $\mathcal{T}$.

Proof Given any bounding function β, we can define two associated bounding functions $\hat{\beta}$ and $\tilde{\beta}$ as follows:

- $\hat{\beta}(T) = \beta(T)$
- If T' is an immediate subtree of T, then $\hat{\beta}(T') = \beta(T') \cap \hat{\beta}(T)$.
- $\tilde{\beta}(L) = \hat{\beta}(L)$ for all leaf nodes of T, and $\tilde{\beta}(T) = \Omega$ for all other nodes.

Then the following lemma holds:

◇ S-bound Distribution Lemma

> Given a bounding function β, then if any of β, $\hat{\beta}$ or $\tilde{\beta}$ is an S-bound function, so are all the others.

To prove the lemma, note that $\hat{\beta} \sqsubseteq \beta$ and $\hat{\beta} \sqsubseteq \tilde{\beta}$, and so if $\hat{\beta}$ is an S-bound function, so are β and $\tilde{\beta}$. Further, if β is an S-bound function then we may apply the Downward Theorem repeatedly to show $\hat{\beta}$ is. So it will suffice to show that $\hat{\beta}$ is an S-bound function if $\tilde{\beta}$ is. Now note that if T has immediate subtrees T_1 and T_2, then if β' is an S-bound function with $\beta'(T_1) = \hat{\beta}(T_1)$ and $\beta'(T_2) = \hat{\beta}(T_2)$, then by applying the Upward Theorem to T we can show that $\beta'[T \vdash \beta'(T) \cap (\beta'(T_1) \cup \beta'(T_2))]$ is an S-bound function, and as $\beta'(T) \cap (\beta'(T_1) \cup \beta'(T_2)) \subseteq \hat{\beta}(T_1) \cup \hat{\beta}(T_2) \subseteq \hat{\beta}(T)$ then we can use induction, starting with $\tilde{\beta}$, to show that $\hat{\beta}$ is an S-hound function. This completes the proof of the lemma.

So if β_1 and β_2 are S-bound functions, so are $\tilde{\beta}_1$ and $\tilde{\beta}_2$. Further, it is easily shown that $\widetilde{\beta_1 \odot \beta_2} = \tilde{\beta}_1 \odot \tilde{\beta}_2$, and so it will suffice to show that either is an S-bound function. Given any $\iota' \sqsubseteq \iota$, and any $\beta^* \sqsupseteq \tilde{\beta}_1 \odot \tilde{\beta}_2$, we need to show that $\mathcal{I}'_{\beta^*}(T) = \mathcal{I}'(T)$. We define β'_1 and β'_2 by $\beta'_i(T) = \beta^*(T) \cup \tilde{\beta}_i(T)$ for all trees T and $i = 1, 2$—then both are S-bound functions, with $\beta^* = \beta'_1 \odot \beta'_2$. Further, define $\iota''(L) = \iota'(L) \cap \beta'_2(L)$ for all leaf nodes L. Thus $\mathcal{I}''_{\beta'_1}(L) = (\iota'(L) \cap \beta'_2(L)) \cap \beta'_1(L) = \mathcal{I}'_{\beta^*}(L)$ for all such leaf nodes. But β'_1 is an S-bound function, and $\iota'' \sqsubseteq \iota$, and so $\mathcal{I}'_{\beta^*}(T) = \mathcal{I}''_{\beta'_1}(T) = \mathcal{I}''(T)$ as both bounding functions return Ω away from leaf nodes. Similarly, we can show that $\mathcal{I}''(T) = \mathcal{I}'_{\beta'_2}(T) = \mathcal{I}'(T)$. Thus $\mathcal{I}'_{\beta^*}(T) = \mathcal{I}'(T)$ for all $\iota' \sqsubseteq \iota$ and all $\beta^* \sqsupseteq \widetilde{\beta_1 \odot \beta_2}$, and so $\widetilde{\beta_1 \odot \beta_2}$, and thus $\beta_1 \odot \beta_2$, are S-bound functions.

◇ Redundancy Theorem

> Let T be a tree with a totally consistent bounding function β. If T is any *positive* subtree of T such that $\mathcal{I}(T) \cap \beta(T) = \emptyset$, then $\beta[T \vdash \emptyset]$ is another totally cousistent bounding function on T.

Proof We need to show that $\mathcal{I}_{\beta'[T\vdash X]}(T) = \mathcal{I}(T)$ for any $\beta' \supseteq \beta$ and any set X. Assume, without loss of generality, that $\beta'(T) = \beta(T)$. By lemma Down2 and the fact that β' is totally consistent

$$\mathcal{I}_{\beta'[T\vdash\emptyset]}(T) \subseteq \mathcal{I}_{\beta'[T\vdash X]}(T) \subseteq \mathcal{I}_{\beta'[T\vdash\Omega]}(T) = \mathcal{I}(T) = \mathcal{I}_{\beta'}(T)$$

and so it will be sufficient to show that $\mathcal{I}_{\beta'[T\vdash\emptyset]}(T) \supseteq \mathcal{I}_{\beta'}(T)$.

Consider a new primitive interpretation, ι', given by $\iota'(P) = \iota(P) - \beta(T)$ for all primitives P. We can show, by structural induction, that

$$\mathcal{I}'_{\beta^*}(T^*) = \mathcal{I}_{\beta^*}(T^*) - \beta(T) \tag{13}$$

for any bounding function β^* and any subtree T^* of T (including T itself). Now $\mathcal{I}'_{\beta'}(T) \subseteq \overline{\beta(T)}$ by (13), and $\mathcal{I}'_{\beta'}(T) \subseteq \beta'(T) = \beta(T)$ by the definition of $\mathcal{I}_{\beta'}$, and so

$$\mathcal{I}'_{\beta'}(T) \subseteq \overline{\beta(T)} \cap \beta(T) = \emptyset$$

Thus $\mathcal{I}'_{\beta'}(T) = \mathcal{I}'_{\beta'[T\vdash\emptyset]}(T)$, and so

$$\mathcal{I}'_{\beta'}(T) = \mathcal{I}'_{\beta'[T\vdash\emptyset]}(T) \tag{14}$$

Thus

$$\begin{aligned}
\mathcal{I}_{\beta'[T\vdash\emptyset]}(T) &\supseteq \mathcal{I}_{\beta'[T\vdash\emptyset]}(T) - \beta(T) \\
&= \mathcal{I}'_{\beta'[T\vdash\emptyset]}(T) \qquad \text{by (13)} \\
&= \mathcal{I}'_{\beta'}(T) \qquad \text{by (14)} \\
&= \mathcal{I}_{\beta'}(T) - \beta(T) \qquad \text{by (13)} \\
&= \mathcal{I}_{\beta'}(T) \qquad \text{as } \mathcal{I}_{\beta'}(T) \cap \beta(T) = \emptyset
\end{aligned}$$

as required.