THE EDINBURGH DESIGNER SYSTEM AS A FRAMEWORK FOR ROBOTICS OR THE DESIGN OF BEHAVIOR

R.J. Popplestone

COINS Technical Report 87-47
May 1987

Abstract

I discuss how the Edinburgh Designer System can be extended and used to support symbolic computation for robotics. I conclude that the Algebra Engine requires to handle temporal constructs, groups and tolerances, that the taxonomy can support *activity modules* and that automatic plan formation would require the creation of a *specialist*.

The Edinburgh Designer System as a Framework for Robotics
The Design of Behavior

1 Introduction

The Edinburgh Designer System (EDS) (Popplestone, 1984b, 1985) provides a coherent set of *inference engines* which operate upon a *common formalism* appropriate to the representation of engineering designs in general. This formalism stems from the work of Barrow (1983), but has been extended in various ways, including to provide for the representation of the evolution of a design. The exploration of the space of possible designs is related to the work of Latombe (1977), although conducted in a Prolog framework. The EDS has been implemented at Edinburgh, and has been mounted on a Sun workstation at U.Mass. My present concern at U.Mass is to define extensions of the EDS needed for Robotics.

In the EDS, a design is specified in terms of *modules*, which are engineering functional units (eg a motor or a keyway or a shaft). Thus modules are *not* necessarily rigid bodies, but may be features of bodies (eg an oilway) or assemblies of bodies (eg a gear-box). The extension of the EDS to deal with plan formation requires that modules be regarded as existing in time, for example the activity of drilling of an oilway or the support of a shaft while a gear is fitted to it.

Barrow specified interactions between modules in terms of connections between *ports* of modules. This formalism is appropriate for his domain (the logical analysis of VLSI designs) because interaction between modules always occurs in a standard way through conductors. However in general engineering, interaction between modules can take place in a rich variety of ways, so that knowledge about such interaction cannot be regarded as a static part of the EDS. Considerable conceptual economy can be achieved by treating the interaction between modules as a module in its own right, called an *interface module*. Such interface modules can be regarded as establishing a relational network or graph upon the *concrete modules*.

In order to describe the formalism and behaviour of the EDS, I will use Prolog terms, typeset in mathematical form, with the Prolog convention that variables begin with capital letters. The EDS represents facts about particular modules which have been postulated to exist in the design with terms which are free of Prolog variables, whereas general facts about classes of modules are represented using Prolog clauses, which will contain Prolog variables.

Modules have *parameters*, which are symbols denoting quantities which are determined at design time, and *variables* which are symbols denoting quantities which may vary while the module is operating. For example a gear has as one of its parameters the pitch radius, and as one of its variables its angular velocity. Interface modules will define constraints upon parameters of the modules connected by the interface. For example the interface module *meshing_gears* equates the pitch line velocities of the two component gears. The parameters and variables of particular modules are *ground level* terms. For example $r\$g1$

may denote the pitch radius of the gear $g1$. In fact module parameters and variables are all *substitution instances* of the Prolog term $V\$M$. If we think of $A\$B$ as being A_B, this notation turns out to be quite in accordance with engineering conventions, for example if $g1$ is a gear and r is its pitch radius, then $r\$g1$ corresponds naturally with r_{g1} which could appear in engineering texts.

Modules are classified in a *taxonomy*, which is part of the formal support of the exploration of possible designs. For example, having decided that $g1$ belongs to the module-class *gear*, a further step of detailing it is to decide that $g1$ belongs to the module-class *spur_gear*. Other important steps in detailing involve deciding on values for parameters, and using one of the engines to determine new relationships.

In the EDS, the representation of shape is supported by a Constructive Solid Geometry (CSG) modeller (Requicha, 1978). The current implementation makes use of the ROBMOD modeller (Cameron, 1984), but an interface to the Noname modeller (Armstrong, 1982) is currently being implemented. It is straightforward to interface the EDS to any CSG modeller which has a well defined textual or procedural interface. Current work at U.Mass includes the development of a ray-tracing modeller which will provide high quality images.

We use the term *inference engine* for any program which applies some body of knowledge relevant to the EDS. Such an engine will typically follow paths in the relational structure placed upon concrete modules by interface modules, and infer new relationships between entities which are at some distance from each other in the original structure, or it may derive new relationships by examining cycles in the structure. The inference engines typically perform forward chaining upon facts about the design: it is necessary to place strict limits on this chaining to avoid an undesirable proliferation of deduced facts.

Thus the RAPT interpreter (Corner, Ambler, and Popplestone, 1983) (Popplestone, Ambler, and Bellos, 1980) is used as an inference engine for deducing module locations from defined spatial relations. Spatial relationships hold between geometric features of modules, which are derived from the primitives which make up the shape. Thus the spatial relationship *fits* may be used to express the relationship between a journal on a shaft and the bush of a plain bearing.

The term "specialist" is used in the EDS context to mean systems which it is intended to provide which provide an integrated treatment of knowledge about some sub-domain of design, for example planning of machining or assembly. specialists will be procedural, but will make use of appropriate declarative knowledge held in the taxonomy.

Any system which supports the activity of design must cope with the fact that designing involves exploring a space of possible designs. The exploratory nature of designing is treated by regarding any statement that the human designer makes to the system as an *assumption*, which may or may not form a part of the final design. For example, a designer may wish to consider the possibility of using an electric motor or a hydraulic motor in a given application, or, at a greater level of detail, the possibility of using a journal bearing versus that of using a rolling element bearing. While some of the design work associated with a given assumption may be lost if an alternative assumption is finally

The Edinburgh Designer System as a Framework for Robotics
The Design of Behavior

1 Introduction

The Edinburgh Designer System (EDS) (Popplestone, 1984b, 1985) provides a coherent set of *inference engines* which operate upon a *common formalism* appropriate to the representation of engineering designs in general. This formalism stems from the work of Barrow (1983), but has been extended in various ways, including to provide for the representation of the evolution of a design. The exploration of the space of possible designs is related to the work of Latombe (1977), although conducted in a Prolog framework. The EDS has been implemented at Edinburgh, and has been mounted on a Sun workstation at U.Mass. My present concern at U.Mass is to define extensions of the EDS needed for Robotics.

In the EDS, a design is specified in terms of *modules*, which are engineering functional units (eg a motor or a keyway or a shaft). Thus modules are *not* necessarily rigid bodies, but may be features of bodies (eg an oilway) or assemblies of bodies (eg a gear-box). The extension of the EDS to deal with plan formation requires that modules be regarded as existing in time, for example the activity of drilling of an oilway or the support of a shaft while a gear is fitted to it.

Barrow specified interactions between modules in terms of connections between *ports* of modules. This formalism is appropriate for his domain (the logical analysis of VLSI designs) because interaction between modules always occurs in a standard way through conductors. However in general engineering, interaction between modules can take place in a rich variety of ways, so that knowledge about such interaction cannot be regarded as a static part of the EDS. Considerable conceptual economy can be achieved by treating the interaction between modules as a module in its own right, called an *interface module*. Such interface modules can be regarded as establishing a relational network or graph upon the *concrete modules*.

In order to describe the formalism and behaviour of the EDS, I will use Prolog terms, typeset in mathematical form, with the Prolog convention that variables begin with capital letters. The EDS represents facts about particular modules which have been postulated to exist in the design with terms which are free of Prolog variables, whereas general facts about classes of modules are represented using Prolog clauses, which will contain Prolog variables.

Modules have *parameters*, which are symbols denoting quantities which are determined at design time, and *variables* which are symbols denoting quantities which may vary while the module is operating. For example a gear has as one of its parameters the pitch radius, and as one of its variables its angular velocity. Interface modules will define constraints upon parameters of the modules connected by the interface. For example the interface module *meshing_gears* equates the pitch line velocities of the two component gears. The parameters and variables of particular modules are *ground level* terms. For example *r$g1*

may denote the pitch radius of the gear $g1$. In fact module parameters and variables are all *substitution instances* of the Prolog term $V\$M$. If we think of $A\$B$ as being A_B, this notation turns out to be quite in accordance with engineering conventions, for example if $g1$ is a gear and r is its pitch radius, then $r\$g1$ corresponds naturally with r_{g1} which could appear in engineering texts.

Modules are classified in a *taxonomy*, which is part of the formal support of the exploration of possible designs. For example, having decided that $g1$ belongs to the module-class *gear*, a further step of detailing it is to decide that $g1$ belongs to the module-class *spur_gear*. Other important steps in detailing involve deciding on values for parameters, and using one of the engines to determine new relationships.

In the EDS, the representation of shape is supported by a Constructive Solid Geometry (CSG) modeller (Requicha, 1978). The current implementation makes use of the ROBMOD modeller (Cameron, 1984), but an interface to the Noname modeller (Armstrong, 1982) is currently being implemented. It is straightforward to interface the EDS to any CSG modeller which has a well defined textual or procedural interface. Current work at U.Mass includes the development of a ray-tracing modeller which will provide high quality images.

We use the term *inference engine* for any program which applies some body of knowledge relevant to the EDS. Such an engine will typically follow paths in the relational structure placed upon concrete modules by interface modules, and infer new relationships between entities which are at some distance from each other in the original structure, or it may derive new relationships by examining cycles in the structure. The inference engines typically perform forward chaining upon facts about the design: it is necessary to place strict limits on this chaining to avoid an undesirable proliferation of deduced facts.

Thus the RAPT interpreter (Corner, Ambler, and Popplestone, 1983) (Popplestone, Ambler, and Bellos, 1980) is used as an inference engine for deducing module locations from defined spatial relations. Spatial relationships hold between geometric features of modules, which are derived from the primitives which make up the shape. Thus the spatial relationship *fits* may be used to express the relationship between a journal on a shaft and the bush of a plain bearing.

The term "specialist" is used in the EDS context to mean systems which it is intended to provide which provide an integrated treatment of knowledge about some sub-domain of design, for example planning of machining or assembly. specialists will be procedural, but will make use of appropriate declarative knowledge held in the taxonomy.

Any system which supports the activity of design must cope with the fact that designing involves exploring a space of possible designs. The exploratory nature of designing is treated by regarding any statement that the human designer makes to the system as an *assumption*, which may or may not form a part of the final design. For example, a designer may wish to consider the possibility of using an electric motor or a hydraulic motor in a given application, or, at a greater level of detail, the possibility of using a journal bearing versus that of using a rolling element bearing. While some of the design work associated with a given assumption may be lost if an alternative assumption is finally

chosen, it is important to avoid the massive loss of effort which would arise if chronological backtracking were employed, and consequently we employ an Assumption Based Truth Maintenance System (ATMS) (deKleer, 1984) to manage these assumptions. As well as assumptions about the existence of particular modules, the design process will also involve making assumptions about the values of parameters of modules and the spatial relations between their features.

Specialists likewise may make use of ATMS to support a more automatic exploration of design space in their area of specialism.

The EDS has been implemented in Poplog (Hardy, 1984) (Sloman and Hardy, 1983) running under Unix on a Sun workstation. While the basic formalism is expressed in Prolog terms, most of the engines described have been written in Pop-11, or a mixture of Pop-11 and Prolog.

2 The Algebra Engine

Of all the inference engines, the Algebra Engine is the most basic. It is capable of simplifying expressions denoting a number of entities important in engineering and of interest to robotics, including locations, vectors and Constructive Solid Geometry shape descriptions. It also embodies Minipress, a system with many of the capabilities of the Press (Bundy, 1979) program for solving equations involving trancendental functions symbolically. Press was developed at Edinburgh by Bundy and his co-workers.

The Algebra Engine was originally conceived of as being an extension of Press. The choice of Press as the basis for the Algebra Engine was determined by the need to solve equations in trancendental functions, since these commonly occur in engineering, and the need for an engine which could be readily interfaced to form a sub-system of the EDS. There are a number of algebraic manipulation packages available on the market, for instance REDUCE and MACSYMA, both of which are LISP based. Both of these packages have a front end which means that users do not have to couch their needs in rebarbative LISP syntax. However they are not readily accessible in a form which makes them ideally suited to be sub-systems of the EDS.

2.1 Simplification

We shall use $T1 \longrightarrow T2$ to mean that the Algebra Engine unconditionally rewrites an instance of term T1 as term T2 during simplification. These rewrite rules do not constitute a complete definition of simplification. The statement $T1 = T2$ means that the Algebra Engine may replace an instance of $T1$ with $T2$, or conversely, depending upon circumstances.

The Algebra Engine can be called upon to simplify a term by executing the Prolog goal $X := Y$ where Y is bound to the term being simplified. For example

$$X := a + b + 2 * a + c - b$$

will bind X to $a*3+c+2$.

Algebraic simplification is not in general a straightforward, or even well defined problem. There is not, in most algebras, a standard "canonical form" into which all expressions can be transformed and which will serve as a basis for doing any operation subsequently required. Thus expression simplifiers for the ordinary algebra over the reals or complex numbers will normally do a number of standard useful operations, leaving others to be performed by specialist code for specialist purposes, or to be specified by options in the simplifier.

The EDS Algebra Engine does not make use of a formal theory of type, but relies upon a use of symbols which is consistent with laws built into the simplifier. Thus, for example it is not possible to use + to mean boolean *or* (or *union*) operation, since the Algebra Engine assumes that + obeys the laws of an abelian group and a boolean algebra does not obey the same laws (eg. $X+Y=X+Z$ does not imply $Y=Z$ if '+' is interpreted as union). In terms of Universal Algebra (Cohn, 1965), we restrict any set of entities upon which function symbols operate to be a *variety*. It should be noted that 0 and 1 are to be regarded as nullary operators.

2.2 Datatypes and basic notational conventions

It is necessary to represent integers, real numbers, 3-vectors, locations (ie Euclidean transformations of 3-space), matrices, and shapes defined as subsets of 3-space. In the present implementation of the EDS integers and reals are representated as fixed and floating point quantities respectively. However rational arithmetic is available within Poplog and has definite advantages for the more theoretical aspects of design.

We use $X->FX$, where FX is a term in X to mean the function which maps X to FX. (cf. $\lambda x.F(x)$). Thus $I->I\uparrow 2$ is the squaring function.

Other datatypes are represented as terms. Thus vectors are terms with the functor vec. In particular, $vec(X,Y,Z)$ is a 3-vector and $vec(0,0,0)\longrightarrow 0$, since zero is known to the simplifier as the identify of the abelian group operator +.

The unit vectors are denoted by the atoms *ii*, *jj* and *kk*, although these are all rewritten as vec(1,0,0) etc., since these symbols are not regarded as basic operators of the Algebra Engine.

We use the term *location* to mean a rigid transformation of 3 space which preserves handedness of axes, ie a member of the *Euclidean Group*. The best canonical representation of constant locations is a matter of some debate: a presentation of the possiblities is to be found in (Brady, Hollerbach, Johnson, Lozano-Perez, and Mason, 1982). At present the EDS uses a 4×3 array representation, derived from RAPT, although a 4×4 matrix representation would have the advantage that locations were no longer special entities.

$X+Y$ means the addition of entities usually added.

$-$ is used for Binary subtraction and unary negation.

Thus

$$X+Y=Y+X,\quad X+0\longrightarrow 0,$$

$$X + (-X) \longrightarrow 0, \quad X + (Y + Z) \longrightarrow (X + Y) + Z.$$

This latter is in accordance with Prolog conventions. Binary minus is rewritten by $X - Y \longrightarrow X + Y * (-1)$. $X + Y$ may denote the sum of reals, complex numbers, matrices, etc.

$X * Y$ means commutative multiplication, eg. of numbers, scalar multiplication of vector by number. * has zero 0 and identity 1, so

$$X * Y = Y * X, \; X * (Y + Z) = X * Y + X * Z, \; X * (Y * Z) \longrightarrow (X * Y) * Z$$

$$X * 0 \longrightarrow 0, \; 0 * X \longrightarrow 0, \; X * 1 \longrightarrow X, \; 1 * X \longrightarrow X$$

The division operator is eliminated (except in the special representation of rational functions discussed below).

$$X/Y \longrightarrow X * Y \uparrow (-1).$$

$X \uparrow Y$ means X to the power Y. $X \uparrow 0 \longrightarrow 1$.

$X@Y$ means the associative, non-commutative multiplication of x by y, for example of locations, permutations, matrices, and the concatenation of sequences. Thus @ plays the same role as the "." operator in Macsyma. @ is also used for locating any entity eg. $block(1, 2, 3)@rot(ii, 0.1)$ means "a block of dimensions 1 2 3 rotated by 0.1 radian about the x-axis".

inv is the inverse of @. $X@inv(X) \longrightarrow X$, $1@X \longrightarrow X$. $X@1 \longrightarrow X$.

2.3 Polynomials and rationals

Terms of the form $poly(X, A_0, A_2 \ldots\ldots A_n)$ (ie. terms of $n+2$ arguments whose functor is *poly* and whose first argument is X) represent the polynomial $A_0 + A_1X + A_2X^2 + \cdots + A_nX^n$). A rational function is represented as the formal quotient of two polynomials. Note that, except for the case of a polynomial of the above form where X is a constant, conversion to and from these forms is not performed by the := predicate, but is invoked by predicates *poly* and *rational*.

2.4 Sequences and iterations.

$M..N$ denotes the finite sequence $[M, M + 1 \ldots N]$, ie. the Prolog list.

$sigma(S, F)$ sums the function F over the sequence S, eg.

$$sigma(1..3, i- > i \uparrow 2) \longrightarrow 1 \uparrow 2 + 2 \uparrow 2 + 3 \uparrow 2 \longrightarrow 14.$$

$pi(S, F)$ similarly takes the product of the function F over the sequence S.

2.5 General operations.

$mod(X)$ is a numeric measure of the size of any entity. Eg. $mod(V)$ is the modulus of the vector V. *mod* obeys over an abelian group the laws $mod(0) \longrightarrow 0$, $mod(X + Y) =< mod(X) + mod(Y)$.

$mod(M, N)$ is the remainder when the integer M is divided by the integer N.

$unit(X)$ is a version of the entity X which is of unit modulus. Eg. $unit(vec(3, 4, 0)) \longrightarrow vec(0.6, 0.8, 0)$

2.6 Locations.

$trans(X, Y, Z)$ denotes a translation by the vector $vec(X, Y, Z)$. $trans(V)$ denotes a translation by the vector V.

$rot(V, T)$ denotes a rotation by an amount T about a vector V.

2.7 Equations

Terms involving equality are simplified depending upon the entities being equated. In particular, any equation over the reals which involves only on e symbolic quantity is given to Minipress to solve, and simplifies to the form $Variable = Constant$. Equalities on terms which have a functor known to be *free* may give rise to equations of the arguments of the terms.

2.8 Ordering relations.

The symbols <, =< etc. are used to mean a total ordering over the reals, following Prolog conventions. Terms involving them are simplified by collecting constants to the right, non-constants to the left. Any inequality involving just constants is evaluated to the Pop-11 true or false.

The following conventions are used for *partial orderings*:- <:= Less than or equal to in a partial ordering, so that it is reflexive, symmetric and transitive.

$X <:= X$. $X <:= Y$ and $Y <=: X$ implies $X = Y$. $X <:= Y$ and $Y <:= Z$ implies $X <:= Z$.

2.9 Lattices and boolean algebras.

A lattice is an algebra with operators \/ and /\, which are associative an commutative and obey the axioms:–

$A/\backslash(A\backslash/B) = A$. and $A\backslash/(A/\backslash B) = A$.

Boolean algebras are special cases of lattices, and in particular the point sets defined by Constructive Solid Geometry form a Boolean Algebra. Boolean algebras are distributive over both \/ and /\, and admit a subtraction operation \, for which

$(A\backslash B)\backslash/B = A$.

It can be shown that lattices are partially ordered unter the relation
$X <:= Y$ iff $A/\backslash Y = X$
It should be noted that lattices do not in general obey the distributive laws of boolean algebra, so that these cannot be built into the Algebra Engine.

2.10 Shapes

Constructive Solid Geometry is used to represent shapes using the following primitives:–
block(X, Y, Z) denotes a cuboid, centroid at the origin, with the stated dimensions along the coordinate axes.
cyl(L, R) denotes a finite cylinder of length L and radius R, centroid at the origin, axis along the Z-axis, and
cone(H, R) denotes a cone of height H and radius R, base centre at the origin, axis along the Z-axis.
sph(R) denotes a sphere of radius R, centre at the origin.
tor$(R1, R2)$ denotes a torus of minor radius $R1$ and major radius $R2$, centroid at the origin, axis of radial symmetry along the z-axis.
These primitives are combined with the boolean operations \/ and /\ for union and intersection, and with \ denoting set subtraction. The term *Shape*@*Loc* denotes a *Shape* relocated by the location *Loc*.
It is a convenience to allow *Shape*@*Vec* to denote the *Shape* translated by the vector *Vec*.
A block has faces called *top, bottom, left, right, front and back*, named under the convention that the X axis is forward, the Y axis is left and the Z axis is upward. These names are preserved under rotation. The lower, upper and curved faces of a cylinder are called the *proximal, distal and curved* faces.
It should be noted that there are some difficulties with the treatment of shape outlined above. A desirable property of any formalism is *referential transparency*, which means that we can always substitute one term for another if the two are equal without changing the sense of what is said. If we regard the CSG primitives as *just* denoting subsets of 3-space, then the face naming conventions, which are needed for specifying spatial relationships, imply a lack of referential transparency. For example if $b1 = block(1, 2, 3)$ and $b2 = block(1, 2, 3)@rot(ii, pi)$ then, as sets, $b1 = b2$, but the *bottom* of $b1$ is the *top* of $b2$. A potential resolution of this difficulty would be to treat shapes not as subsets of space, but as functions from 3-space to a set of *labels*. It is possible to ensure that such *labelled sets* form a boolean algebra by requiring the labels themselves to form a boolean algebra.

3 Bodies

An important property of a module is that of being a *rigid body*, which implies that the nominal relative positions of any sub-modules are determined at design time. The shape of a body module m is determined from the shape of the modules listed in the parameter

parts$M$, but the way that this shape is computed may depend upon the module-class to which M belongs. In what follows, I shall use "body" to mean a module which happens to be a rigid body, and use "features" loosely to mean a sub-module, or geometric features of the shape of the body.

It should be noted that, during the course of design, the existence of many modules may be established long before their membership of bodies is established, and indeed body membership may change during the course of design. For example, it may be seen to be necessary to make a module such as a valve seat, which requires fine grinding, as an insert rather than be machined as an interior feature of a larger body.

3.1 Group Theory

Many modules will have symmetry, either actual or functional. It is important to extend the EDS to have an explicit representation of these symmetries. In Popplestone (1984a) it is shown that the capabilities of RAPT inference engine could be generalised if a group theoretic representation were used, and in particular an exploitation of finite symmetries would be possible.

The basic idea is that any spatial relationship between features determines the relative positions of bodies possessing these features *modulo the symmetry group(s) of these features.* Suppose body $B1$ has feature $F1$, and body $B2$ has feature $F2$, then one possible implication of the statement that $F1$ fits $F2$ (or $F1$ fits $F2$ exactly) is that they have the same symmetry group (up to automorphism), and that the relative location of $B2$ with respect to $B1$ is a coset of that symmetry group. For example, if the socket of a socket wrench has a six fold symmetry (some do, others have twelve-fold symmetry), then its symmetry group is the cyclic group we shall denote by $cy(6)$. To be more precise, in our context $cy(6)$ denotes a particular cyclic subgroup of the euclidean group. If the socket fits a bolt head with six fold symmetry, then the position of the wrench with respect to the bolt is determined to be a member of a coset of the common symmetry group, that is $T1@cy(6)@T2$ where $T1$ and $T2$ are constant transformations. (eg. if the axes are centrally embedded in the bodies, then perhaps $T = trans(0,0,6)$) and $T2 = 1$).

From Popplestone (1984a) we see that a basic operation which needs to be performed on groups is the simplification of expressions involving intersections of cosets of the Euclidean group. It is also possible that a representation of groups of permutations could be desirable in treating sets of identical entities.

A suitable notation for these groups needs to be chosen, and a suitable representation for performing the necessary computations. A classification of the infinite groups can be found in Hervé (1978). Many of the finite or finitely generated sub-groups of interest are crystallographic groups (Shokichi, Iyanaga, and Yukiyosi, 1968). A uniform representation of the finitely generated sub-groups is possible simply by listing their generators, preferably in a canonical order. The infinitely generated groups can be specified by defining their generators parametrically. A suitable convention would be to denote groups by gp(List), where List is a list of functions or constants. eg $gp([t- > rot(kk,t)])$ would denote the

group of rotations about a single axis.

Calculating the intersections of cosets of such sub-groups from first principles involves the solution of equations, which will either be non-linear equations in real variables, or diophantine equations, or both. The implementation of RAPT described in Corner et al (1983) does in fact deal with the former type of equations, in the form of *location equations*. However, solving such problems from first principles is not efficient — it is better to have a catalog of the solutions to commonly occurring ones eg.

$$cy(N)/\backslash cy(M) \longrightarrow cy(hcf(N,M)).$$

ie. the intersection of two cyclic sub-groups is the cyclic sub-group whose order is the hcf of the orders of the two sub-groups. Following Knuth, Bendix, Huet (Huet, 1978), it is possible to check such rewrite rules for *confluence*, and add new rules, depending upon the existence of "critical pairs". Thus if we say:

$$gp([rot(kk,pi/N)])/\backslash gp([t- > rot(kk,t]) \longrightarrow gp([rot(kk,pi/N)])$$

ie. the intersection of a cyclic subgroup with the sub-group of rotations about the z-axis is the cyclic subgroup, we are creating a need for rewrite rules to convert from the form $cy(N)$ to $gp([rot(kk,pi/N)])$. It is important when producing a final form of a coset expression to choose a standard member of the automorphism class of any sub-group occurring — for example, where a single axis of rotation is involved, that should be about the z-axis.

4 Exploring Design Space

The EDS is not intended to perform fully automatic design: it is a *Designer's Apprentice*, which is capable of performing some Design functions automatically, but is dependent upon human guidance for the more strategic decisions. The EDS is also seen as a vehicle upon which more fully automatic design *specialists* can be built. It will play this role also in Plan Formation.

The process of design is the construction of a design description document (DDD), which is held as consequences within the ATMS. The DDD contains a set of assumptions, and a set of consequences (called *values* by de Kleer). Each assumption corresponds to a design decision. For example, the assumption $g1$: *gear* says that $g1$ is a gear, and the assumption $r\$g1 = 10$ says that the parameter $r\$g1$ has the value 10. There is no requirement for assumptions to be consistent. For example, the assumption $r\$g1 = 20$ might coexist with the assumption that $r\$g1 = 10$. However, a final design is characterised by a set of assumptions which must not be known by the system to be inconsistent. So the two values for $r\$g1$ quoted above can only form part of alternative possible designs.

Consequences are held as Pop-11 records of the form $conseq(N, B, As, J, P)$ where B is a term with a truth functional value (eg. $r\$g1 = 10$) and As is the set of assumptions that the consequence depends on. P, the parents, is the set of consequences that the

consequence was derived from. J is the justification --- the name of the rule used to derive the consequence.

While a small design exercise could be essayed with no additional structure on the ATMS, serious design requires us to provide the concept of *focus*, whereby a limited set of consequences is in use for forward chaining, corresponding to a particular part of the design that is being considered. For example, in the case of a gearbox design, the initial focus will be on the gear teeth, followed by the location of the bearings on the shafts, the loading of these bearings, the analysis of the shafts for stiffness and strength, the detailing of bearings, housings and gear wheels, etc. An experimental focussing capability is available in the EDS, which allows the user to select particular consequences, or consequences containing particular variables, or which depend on particular assumptions.

The Prolog goal *assume(Term)* is used to enter assumptions into the ATMS. For example, : $-$*assume*(*g*1 : *spur_gear*) states that the module *g*1 is a spur gear.

Let us now discuss the interaction between the ATMS and Prolog. When a new fact is entered in the ATMS it can give rise to forward chaining. Possible forward chainings are defined by the prolog predicate *implies*, and by separately encoded equality propagation. The result of a forward chaining is always given to the Algebra Engine to simplify before it is entered in the DDD.

4.1 Equality propagation.

Only consequences of the form *Variable* = *Constant* give rise automatically to forward chaining. Any consequence of which contains *Variable* spawns another consequence with *Constant* substituted for it. If the new consequence has the same assumption base as the old one, the old consequence is said to be subsumed by the new one, and is deleted. New consequences are simplified by the Algebra Engine, and may give rise to yet more equality substitutions. Recall that the Algebra Engine may invoke Minipress to solve equations during this simplification process, which will thus treat systems of linear equations if their matrix is triangular. A separate engine is being provided to solve non-triangular systems of linear equations.

4.2 Propagation of other consequences

Propagation of other consequences is determined by the predicate *implies*. Implies is defined both for single consequences and for pairs of consequences. Every new consequence is given to *implies*, and is paired up with old consequences and the pair is given to *implies*. For example consequences of the form M : *Mc* state that M is a member of the module class *Mc*. Thus *g*1 : *gear* states that *g*1 is a *gear*. *implies* deals with this by *consulting* a file containing the definition of the module class (if necessary) and instantiating all of the constraint clauses. This forward chaining can proceed quite far, since a module may have *parts* each of which is stated to be a member of some other module class. However the chaining is normally terminated before the whole design is automatically expanded into

the DDD by the fact that some modules will be at too abstract a level in the taxonomy to have parts specified.

5 The Taxonomy

Module class definitions are held in an external form in the taxonomy, and translated by a program called Dracula into prolog clauses which are held textually in files. The external form contains parameter and variable declarations, tagged pieces of English intended for appropriate regurgitation to the user, constraint and table definitions. One view of the taxonomy is that it constitutes something like a generative grammar for designs.

All properties of modules (eg. names and types of variables and parameters) are strictly inherited down the taxonomy, which puts significant consistency constraints upon that structure. Thus we should be in no danger of finding off-white elephants in our trees (Brachman, 1985).

Currently the structure of the taxonomy is implemented by textual references in *header files* associated with each module class. This structure is very cumbersome to modify interactively, making it difficult to install new modules, and it is intended to change it.

5.1 Constraints and Tables.

Constraints are relationships between the mathematical entities which characterise a module or modules. The general form of a constraint as a Prolog clause is:–

$$constraint(Mc, M, B, Id).$$

where Mc is the module class to which the constraint applies, B is a boolean term expressing the relationship, and Id is an identifier used to refer to the constraint, so that it can be identified and used by the appropriate inference engines. (Note the EDS has at present an extra place in the *constraint* predicate associated with the implementation of ports, which was at first essayed). M is a prolog variable which occurs in B and which is bound to give an instance of the constraint which will be entered into the DDD.

For example

$$constraint(spur\_gear, G, ft\$G = tor\$G/r\$G, 1)$$

says that the tangential force on a spur gear is equal to the torque divided by the pitch radius. If we say, for example, $g1 : spur\_gear$, then the equation $ft\$g1 = tor\$g1/r\$G1$ is loaded into the DDD as a consequence.

Shapes are defined by means of shape constraints, which have the same form as any other equations. The shapes associated with modules may be positive, that is indicating the presence of material, or negative, that is indicating its absence. The parameter $shape\$M$ is used to denote the nominal positive shape of any module. A module such as an oil-way will have a negative shape, indicating material that must not be present, and a positive shape, indicating material that must be present to provide adequate walls.

In engineering, as well as relationships between parameters an variables being expressed as equations, a tabular or graphical form is often used, either because the mathematical form is felt to be too complex for hand computation, or because the relationship has been established empirically, or to express some arbitary convention. For example, the factor by which the power capacity of a pair of gears is reduced with increasing speed is expressed in the form of a graph, and is input to the EDS as a table.

Thus the EDS is capable of handling constraints in tabular form, which requires a descriptive apparatus for defining what the entries in a table mean. We have used the work of Codd (1970) (see also Popplestone, 1979) as a guide to this, although there is a need to extend the concepts of relational databases because they are dependent on the idea of an exact match between components of tuples, whereas some engineering tables require an interpolation to be done. While a detailed account of the treatment of tables is outwith the scope of this paper, suffice it to say that tables are similar to Codd relations with module parameters or variables as column headers, and that they undergo manipulations corresponding to those undergone by constraints, with the relational join operation playing a similar role to the elimination of a variable or parameter between two constraints.

6 Invoking the Inference Engines

Many inference engines will have been written with representation conventions other than those used in the EDS, or indeed may be completely separate programs. Thus their use will in general involve a measure of data conversion.

6.1 Invoking Press

Press (Bundy, Byrd, Luger, Mellish, and Palmer, 1979) is a system which is able to solve symbolic equations, and is used to allow the equations arising in design to be solved for an appropriate variable or parameter, for example to allow it to be eliminated between two equations. Press is written in Prolog, with a compatible representation of terms to that used in the EDS, so that the interfacing task is straightforward. Press itself is however a very large and cumbersome program. The purposes of the EDS are adequately met by Minipress, which makes use of Bundy's methods, and some of the axioms, but is a new program written by the author.

The simplifier automatically makes use of Minipress if it discovers an equation with only one symbolic quantity — the equation is solved and the result substituted for.

Press will be available to the user of the EDS to allow him to solve a named equation (occurring as a consequence) for a named variable or parameter. More complex use is envisaged to allow other inference engines, or specialists, which will trace through relational paths, to eliminate variables of a certain class from equations of a certain class at each step. For example, the input torque to a drive train can be deduced from the output torque by equating the input and output torques of successive stages and applying the torque conversion laws appropriate to each stage.

6.2 Invoking the RAPT inference engine

In the EDS, rigid modules are the conterpart to bodies in RAPT. In any focus of the ATMS, certain spatial relations will be known to hold between features of modules. The RAPT inference engine can be used to derive explicitly more constraining relations from these, and in some cases to infer fixed relative locations for the modules. Features are named by reference to the conventional names of features of the primitives of module shapes.

RAPT is written in POP-2 (Burstall, Collins, and Popplestone, 1971) and so is readily available within the EDS, its invocation requires the building of data structures encoding the relationships known to the EDS, and expressing the locations of module features. These latter are inferred by examining the CSG definition of the module shapes.

6.3 Invoking the Modeller

The modeller exists as a separate process, and communication with it is via some form of Inter Process Communication. The elementary interface which has at present been implemented simply makes use of a printing routine which prints CSG data structures in the syntax required for input to the modeller, and sends the resulting character stream to the modeller through a Unix pipe, and listens through another pipe for the modellers's response. This interface is used both to request the modeller to draw shapes and to query the modeller about matters such as the volume of shapes and whether they intersect. The CSG trees thus transmitted are variable free, so than no attempt is made to use the modeller's capability to handle symbolic descriptions.

More efficient use of the modeller would allow the EDS to to feed variable-containing shape expressions to it and to keep a tally of the values the variables are bound to. Also a procedural interface rather than a textual one may be used.

7 Plan Formation

The approach to plan-formation which I am advocating in this paper is to construct plans as *behaviour modules* in the EDS. The kernel of any element of behaviour arises from the existence of modules in the design itself. If a shaft module *sh*1 exists in the design, then the activity *acquire*$*sh*1 can be generated. Whether the acquisition involves buying or manufacturing is not immediately manifest: choosing the acquisition method is a matter of descending the taxonomy. If the "buy" choice is made, the acquisition module will have parameters like *supplier*$*sh*1 and *cost*$*sh*1, whereas if the "make" choice is made, the acquisition module will become a complex structure where decisions about the formation of various sub-modules are made, for example bearing seatings and gear hub seatings. Potential strategies for forming these will themselves be found in the taxonomy, and will be taxonomically structured as in (Tenenberg, 1986).

Likewise the existence of an interface module in which a gear drives a shaft implies (if they are separate bodies) the existence of an activity of assembling the gear to the shaft.

However, the representation of activities does require an extension of the symbolism of the present EDS. For example, we need to be able to say that one activity occurs before another, or during another (Allen, 1983).

The temporal ordering of activities in assembly is strongly dependent upon geometry. It is, of course, important to avoid considering in detail all of the possible assembly sequences, and it is certainly worthwhile analysing any activity module both for its feasibility in isolation, and to examine the temporal dependencies of activities taken pairwise. Koutsou (1986) has shown how it is possible to plan trajectories for isolated bodies using two key components of the EDS, the Spatial Relation Engine, RAPT, and the Modeller. This approach can be used to answer questions such as "Suppose I insert shaft *sh*2 in bearing *b*1 before I insert bearing *b*2 in bearing housing *bh*2, can I plan a suitable trajectory for this latter insertion?", thereby obtaining facts about the possible ordering of assembly operations, even if these are not necessarily completely adequate for determining the order. Developing an ordering in this way offers the possibility of treating what the planning literature refers to as "goal interaction", which is particularly difficult in robotics if we are dealing with bodies having a realistic range of shapes because it is not possible to codify the interactions as can be done in the "blocks world".

It is also necessary to treat what are referred to in the planning literature as "pre-conditions" and "postconditions"(or "effects"). The most important post-condition of an activity module *acquire*$*M*, where *M* is a module which is present in the final design, is simply that *M* exists, and normally this will be, from the point of view of the EDS, a permanent state of the world, although of course trial assemblies are not unknown. However it will also be necessary to design jigs and fixtures for manufacture, and it is possible that some of these will have to be assembled and disassembled during the course of manufacture. It will be natural to treat the design of a jig or fixture with the same EDS mechanisms as any other design: hence it will be necessary to admit that the main effect of an activity may have a finite future.

The preconditions of an activity can be treated as a requirement for the existence of certain interface modules. For example, suppose bearing *bg*1 is to be inserted in a bearing-housing *bh*1 belonging to an end-plate *frame*1 by an activity *acquire*$[*bg*1, *bh*1]. Let *frame*1_*jigged* be an interface module in which *frame*1 is mounted in a jig. Let *bg*1_*gripped* be an interface module in which *bg*1 is gripped by a robot gripper. Then

$$acquire\$[bg1, bh1]\ during\ frame1\_jigged$$

holds, as does

$$bg\_gripped\ overlaps\ acquire\$[bg1, bh1]$$

. It should be noted that in the realm of assembly planning and machining planning the concept of *module identification* (qv.) plays an important role. For example if we are going to perform another insertion into a feature of *frame*1 the interface module *frame*1_*jigged*_1 to exist, then it is probably more efficient to make the identification *frame*1_*jigged* = *frame*1_*jigged*_1 if possible, ie. to use just the one jigging operation.

Of course, the result of this identification is that conditions upon the two jigged states are conjoined, and may become unsatisfiable. The automatic generation of such identifications will be the task of *assembly specialists* which will cluster operations for similarity in such properties as direction of approach, so that there will be a strong expectation of satisfiablity of conjoined preconditions over these clusters.

8 User Interface

The present implementation of the EDS has a minimal user interface, requiring the user to input assumptions into the ATMS with the *assume* predicate. Some work has been done in displaying the taxonomy as a visible tree. It is anticipated that a more user friendly system will make use of modern interactive graphical capabilities, where for example an assumption about the existence of a module will be generated by pointing to a token for the module class presented on the display. Alphanumeric input will still be required for other purposes, for example to override the default name assigned by the system to a new module, or to assume a value for a parameter, but we would use the technique familiar in the Mackintosh, where the significance of such input is determined by its location in a box on the screen.

Considerable thought will have to be given to the presentation to the user of the current design state, since an undifferentiated mess of equations and drawings will be of little value to him. This will almost certainly have to make use of a windowing structure in the work station, with data that is in some sense related grouped together in the same window. Some preliminary work to these ends has already been done.

While the Prolog form of input and output of mathematical expressions is considerably more user friendly that that of Lisp, we expect to implement further improvements in typography by providint for the display of greed letters and subscripts and superscripts.

9 Implementation

The EDS has been implemented in a demonstrable form, and has been mounted on Sun workstations at various industrial and academic sites of the Design to Product Demonstrator, and at UMass. Throughout this paper I have distinguished between what has been done and what might be done or is planned to be done by using suitable modal verbs for the latter.

10 Discussion

It should not be assumed that the definition of a design in terms of modules will result in a modular design, since the modules referred to in this paper are basic engineering entities, which can be freely combined. In order to permit the system to contain definitions of

fairly complex high level modules, while not coercing a modular form of design, a specific capability, that of *modular identification* needs to be introduced. This allows the designer to decide that two modules which were originally considered to be distinct are in fact identical. For example, an electric motor always has a frame, which supports the working parts, and a transmission likewise always has a frame. However in an optimised design the frame of the motor and of the transmission may be identified.

A number of important issues have not been discussed in this paper. All of the locations and shapes of bodies have been treated as though knowledge of the *nominal location* and *nominal shape* were adequate. Some excellent work has been done on *reasoning with uncertainty*, for example Brooks (1982) has adopted an algebraic approach to treating the problem of errors in robot planning. Requicha (1983) provides an interpretation of engineers' tolerancing of shape in a CSG context. Fleming (1985) has treated the accumulation of location tolerance arising from loose fits, and is studying the accumulation arising from toleranced bodies. Lozano-Perez, Mason and Taylor (1984) have studied how a goal can be achieved despite position uncertainty provided the uncertainty envelope lies in a *pre-image* of the goal.

Providing a treatment of tolerance in the EDS promises to be a formidable activity. However I would suggest the following guide-lines.

1. The development of a formalism for specifying tolerance on scalar parameters, and the provision of a tolerance inference engine to propagate tolerances through constraints.

2. The examination of Fleming's techniques for representing location tolerances, with a view to their possible adoption. Fleming uses location tolerance zones which are intervals of the angles, but have a linear relationship between angular tolerance and linear dimensions. These structures, while they may not give as tight a hold on tolerance as is obtainable in theory by Brooks' approach, can be manipulated readily without heavily taxing the symbolic computational capabilities of a system.

3. The adoption of Requicha's scheme for tolerancing shapes. This is likely to mean that the *block* primitive falls out of use, since its sides will need to be independently toleranced. It will be replaced by half spaces.

4. Much knowledge about tolerances will be encoded in modules, rather than derived from first principles, in line with the philosophy of the EDS.

Within the framework of the EDS, simple sensors can be treated as modules which bind the values of certain variables. Dealing with complex sensors, such as an image acquisition system with its associated image segmenting software, could be done by a spacialist, since the EDS does contain constraints upon the scene, and in the ATMS a system for supporting alternative interpretations.

Acknowledgements

The implementation of the EDS forms part of the Alvey Demonstrator Project "Design to Product", funded in part by H.M. Government through her Department of Trade and Industry.

The EDS is an organic part of the Demonstrator, and owes much to the other collaborators GEC, Lucas, Leeds University, Loughborough University and the National Engineering Laboratory. Individuals who have made important contributions to the evolution of the EDS include A.P.Ambler, T.Arai, P.Davey, I.Powell, M.Sabin and J.Streeter. The EDS was realised by the Edinburgh Team — J.Corney, O.Franks, R.Gray, G.Sahar, S.Renton, T.Smithers and S.Todd.

References

Allen, J.F. 1983. Maintaining Knowledge about Temporal Intervals. *Communications of the Association for Computer Machinery*, Vol 26, No 11.

Armstrong et al 1982. *Noname Description and Users Manual.* Department of Mechanical Engineering, Leeds University, England.

Barrow, H.G. 1983. Proving the Correctness of Digital Hardware Designs. *Proceedings of the National Conference on Artificial Intelligence*, Washington, DC.

Brachman R.J. 1985. "I Lied about the Trees", or Defaults and Definitions in Knowledge Representation. *A.I.Magazine*, Vol 6, No 3.

Brady, M., Hollerbach, J., Johnson, T., Lozano-Perez, T., and Mason, M. 1982. *Robot Motion: Planning and Control.* Cambridge, MA and London: MIT Press.

Brooks, R.A. 1982. Symbolic Error Analysis and Robot Planning. *International Journal of Robotics Research*, Vol 1, No 3, 29-68.

Bundy, A., Byrd, L., Luger, G., Mellish, C. and Palmer, M. 1979. *Solving Mechanics Problems using Meta-level Inference.* In Specialist Systems in the Micro Electronics Age, D. Michie (ed.), Edinburgh: Edinburgh University Press.

Burstall, R.M., Collins, J.S. and Popplestone, R.J. 1971. *Programming in POP-2.* Edinburgh: Edinburgh University Press.

Cameron, S.A. 1984. Modelling Solids in Motion. Ph.D. Thesis, Department of Artificial Intelligence, Edinburgh University.

Clocksin, W. and Mellish, C. 1981. *Programming in Prolog.* New York: Springer Verlag.

Codd, E.F. 1970. A Relational Model of Data for Large Shared Data Banks. *Communi-*

cation of the Association for Computer Machinery 13, 377-387.

Cohn, P.M. 1965. *Universal Algebra.* Jointly published, New York: Harper and Row; Evanston, London, and Tokyo: John Weatherhill.

Corner, D.F., Ambler, A.P., and Popplestone, R.J. 1983. Reasoning about the Spatial Relationships Derived from a RAPT Program for Describing Assembly by Robot. *Proceedings 8th International Joint Conference on AI*, Karlsruhe, DBR.

de Kleer, J. 1984. Choices without Backtracking. *Proceedings Conference American Association for AI.*

Fleming, A. 1985. Analysis of Uncertainties in a Structure of Parts. *International Joint Conference on AI 85*, 1113-1115.

Hardy, S. 1984. A New Software Environment for List-Processing and Logic Programming. In *Artificial Intelligence, Tools, Techniques and Applications*, O'Shea and Eisenstadt (eds.), New York: Harper and Row.

Hervé, J.M. 1978. Analyse Structurelle des Mécanismes par Groupe des Déplacements. *Mechanism and Machine Theory*, Vol 14, No 4.

Huet, G. 1978. *Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems.* Roquenfort, France: IRIA.

Shokichi, Iyanaga and Yukiyosi, Kawada. 1968. *The Encyclopaedic Dictionary of Mathematics, English edition.* Cambridge, MA and London: MIT Press; Tokyo: Iwanami Shoten (Japanese original).

Koutsou, A. 1986. Phd Thesis, Department of Artificial Intelligence, Edinburgh University.

Latombe, J.C. 1977. Une Application de L'Intelligence Artificielle à la Conception Assisstée par Ordinateur (Tropic). Thèse d'état, L'Université Scientifique et Médicale de Grenoble.

Lozano-Perez, T., Mason, M.T., and Taylor, R.H. 1984. Automatic Synthesis of Fine Motion Strategies for Robots. *International Journal of Robotics Research*, Vol 3, No 1, 3-24.

Popplestone, R.J. 1979. Relational Programming. In *Machine Intelligence 9*, D. Mitchie (ed.), Chichester, Sussex: Ellis Horwood.

Popplestone, R.J., Ambler, A.P., and Bellos, I. 1980. An Interpreter for a Language for

Describing Assemblies. *Artificial Intelligence*, Vol 14, No 1, 79-107.

Popplestone, R.J., 1984. Group Theory and Robotics. *Robotics Research: The First International Symposium*, Brady and Paul (eds.), Cambridge, MA and London: MIT Press.

Popplestone, R.J. 1984. The Application of Artificial Intelligence Techniques to Design Systems. *International Symposium on Design and Synthesis*, Japan Socy. Precision Engineering, Tokyo.

Popplestone, R.J. 1985. An Integrated Design System for Engineering. *Preprints of the 3rd International Symposium on Robotics Research*, Gouvieux, France.

Requicha, A.A.G. and Tilove, R.B. 1978. Mathematical Foundations of Constructive Solid Geometry: General Tobology of Closed Regular Sets. TM27A, Production Automation Project, University of Rochester, Rochester N.Y.

Requicha, A.A.G. and Tilove, R.B. 1983. Toward a Theory of Geometric Tolerancing, *International Journal of Robotics Research*, Vol 2, No 4.

Sloman, A. and Hardy, S. 1983. POPLOG: A Multi-purpose Multi Language Program Development Environment. *Artificial Intelligence and Simulation of Behavior Quarterly* No 47.

Tenenberg, J. 1986. Planning with Abstraction. *Proceedings Conference American Association for AI-86*, 76-80.

R.J. Popplestone — A short biography.

Robin J. Popplestone (Professor) Obtained his B.Sc in 1960 from Queen's University Belfast, and spent most of the subsequent 25 years working on Artificial Intelligence and Robotics at the University of Edinburgh. In the 60's, with R.M. Burstall, he developed the POP-2 language which combined the semantics of Lisp with an ALGOL-like syntax and a richer variety of datatypes. In the 70's he worked on robotics and vision, including the demonstration of robotic assembly using 2-D vision and touch, the use of structured light for 3-D sensing and the specification of assembly in terms of spatial relations. Before leaving Edinburgh to come to UMASS (in 1986) he was the architect of the Edinburgh Designer System (EDS), which applies AI techniques to the support of engineering design at multiple conceptual levels. At UMASS he is Co-Director of the Laboratory for Perceptual Robotics, where he is concerned with using systems such as the EDS to develop intelligent robotics, and with applying the many computational skills in the UMASS environment to robotics.