A definitive programming approach to the implementation of CAD software

Meurig Beynon
Alan Cartwright

Departments of Computer Science and Engineering
University of Warwick
Coventry CV4 7AL

Abstract

This paper outlines an approach to the implementation of CAD systems that makes use of a programming paradigm based upon definitions ("definitive programming").

It departs from previous research on "pure definitive notations" - special-purpose notations for interaction - and proposes a general-purpose programming model based upon definitive principles. This model is examined as a possible basis for the development of an integrated framework within which to address the broader issues of a design support environment, including constraint handling and user-interface management. This gives a new perspective on the use of definitive principles for interaction in which the emphasis is upon interpreting a family of definitions as one of many possible "intelligent views" of an interactive system. It also establishes a closer relationship between the definitive programming approach to CAD and the study of CAD from an AI perspective than was previously evident.

The design of an appropriate definitive notation for geometric modelling is a fundamental aspect of the application of definitive principles to CAD software. An appropriate basis for such a notation is presented in an Appendix.

A definitive programming approach to the implementation of CAD software

Meurig Beynon
Alan Cartwright
Departments of Computer Science and Engineering
University of Warwick
Coventry CV4 7AL

Keywords: computer-aided design, human-computer interaction, user-interface management, geometric modelling

This paper - a sequel to [2] - outlines an approach to the implementation of CAD systems that makes use of a programming paradigm based upon definitions ("definitive programming"). There are two principal aspects to this research:

(a) the design of an appropriate definitive notation for dealing with the complex geometric problems that are centrally important in CAD applications;

(b) the development of an integrated framework within which to address the broader issues of a design support environment.

The paper is in two parts: an extended account - written by the principal author - of the research so far carried out on (b), and a jointly authored Appendix providing further details of (a).

Of the two aspects, the first is a direct extension of previous work on pure definitive notations for interactive graphics [2]. In contrast, the work that has been done under (b) - though consistent with the view of "dialogue over a definitive notation" as an appropriate intermediate code for user-computer interaction (cf [2]) - involves a significant development of the definitive programming paradigm, and introduces considerations far beyond the scope of [2]. These lead in particular to a new perspective on the use of definitive principles for interaction in which the emphasis is upon interpreting a family of definitions as one of many possible "intelligent views" of an interactive system. This appears to establish a much closer relationship between the definitive programming approach to CAD (cf §5) and the study of CAD from an AI perspective (cf [17]) than was previously evident.

The concept of using definitions as a basis for interactive software was advocated in [1], and is illustrated in the design of the ARCA and DoNaLD graphics systems [3,4]. The merits of using definitive principles as a basis for software for interactive graphics are described in detail in [2]. In particular, the advantages in respect of data representation and abstraction over traditional procedural or purely declarative programming paradigms are explained. It would be misleading to suggest that the application of these principles to the design of graphics for a CAD system is trivial however. This is illustrated by the design of an appropriate definitive notation for geometric modelling, as outlined in §1.2. The problems encountered in this generalisation have such interest and relevance for the entire project as to merit supplementary consideration in an Appendix.

The framework described in [2] is also limited in several other respects. A key idea behind definitive notations is that the current status of a user-computer interaction ("the state of dialogue") is effectively represented by a system of definitions resembling in essence the system of functional relationships between scalar quantities underlying a naive spreadsheet. Such a view of interaction is oriented towards a passive use of the computer in which the responsibility for changing the state of dialogue rests upon the user. In practice, there are many important issues in the design and implementation of CAD systems that demand a more general framework. Valuable though the passive use of sophisticated data description is in the early stages of the design process [20], there is also a very significant role for interaction in which the computer plays an active part (cf [21] p8). Traditional user-interface management issues come to mind in this context [12,14]: the animation of a dialogue through appropriate use of windows, menus, graphical displays, and the use of analogue rather than textual input. Of equal importance are issues such as the monitoring and maintenance of constraints [8,18,22,25]; an activity in which the computer must itself participate in

changing the state of dialogue.

The purpose of this paper is to argue that the essential concept of definitive programming (viz the representation of a state of dialogue by means of a set of definitions that is transparent to the user) can be developed to support the broader concerns raised by the implementation of sophisticated CAD systems. Some justification for regarding CAD systems based upon definitive principles as well-suited for the implementation of "intelligent" CAD software is also given. Indeed, the thesis that computer-aided design systems can be very effectively implemented using definitive principles in such a way that the user has a powerful abstract view of the stages of the interactive design process itself suggests a possible interpretation of "intelligent" computer aided "design".

The paper is divided into 5 sections.

§1 reviews some of the principal ideas discussed in [2] and gives a brief outline of a definitive notation suitable for geometric modelling combining some of the characteristics of both the ARCA and DoNaLD notations. The aim is to illustrate explicitly how the ideas in [2] can be applied to the description of geometric objects in many semantically different ways: as abstract complexes of simplices, as frames comprising finite configurations of simplicial elements, and as realisations of such frames as geometric objects defined by sets of points in space (cf [9,26]). In effect, the state of dialogue over such a notation is interpreted as a formal representation of the user's current model of a geometric object, and the design process is viewed as a process of refinement of this model associated with a sequnce of transitions through different dialogue states.

§2 focusses on some limitations of this "pure definitive notation paradigm" for design as a basis for developing CAD system software.Two principal issues are considered

1) the need in general for a framework of constraints within which the design dialogue must operate (§3),

2) the relationship between the semantics of the geometric design dialogue and the mechanics of the interaction between the user and the computer (§4).

Where 2) is concerned, it may be seen that no provision is made within the pure definitive notation design paradigm for representation of the current state of the device supporting the user-computer interaction. This is in some sense appropriate, in that the abstract design process has meaning independent of the ephemeral state of the screen during some specific design transaction; on the other hand, there is in general a complex interrelationship between the underlying semantics and the current state of the display interface. A major step towards the solution of this problem is the introduction of an auxiliary definitive notation within which to specify the screen display at any stage. The form of the user interface can then be specified as a set of definitions, either by the system or by the user, and subsequently driven by dialogue actions initiated by the computer.

An underlying principle of definitive programming, relevant to both 1) and 2), is that the way in which a transition between dialogue states is effected is not of primary significance; what is crucially important is that the current state of dialogue can be subtly represented, and is at all times transparent to the user. To exploit this principle fully, a richer programming paradigm is required, in which - in the conceptually simplest framework - several processes may participate in the design dialogue (see [5], and compare [16]). To this end, an extended form of definitive programming is described, incorporating definitions and user-defined functions (as in a pure definitive notation) together with actions that - in an appropriate dialogue state - are invoked to change this state (§2.2). §'s 3 and 4 are concerned with the way in which such a computational framework may be used to handle constraints and aspects of the user-interface management in a unified manner, and indicate some of the anticipated advantages of a consistent use of definitive principles.

The final section of the paper discusses the potential merits of a definitive programming approach to the development of an "intelligent" CAD system. A central theme is that the interpretation of a family of variable definitions as an "intelligent view" of an interactive system is a powerful abstraction implicit in many "intelligent" CAD systems based on more traditional paradigms.

§1. Applying definitive principles to CAD

In this section, the basic principles of definitive programming are briefly reviewed, and the design of a definitive notation that has been developed for geometric modelling is outlined.

1.1 Basic principles

A general account of the merits of using definitive principles for interactive graphics is given in [2], to which the interested reader is referred for further background. For convenience, a brief review of the basic concepts and terminology will be given here.

A **definitive notation** is specified by an underlying algebra comprising a set Δ of data types and a family Σ of operators that take the form of pure functions mapping between the data types. The underlying algebra is complemented by variables, whose types are in Δ, and whose values can be defined by algebraic expressions in terms of other variables and explicit values via the operators in Σ. Programming over a definitive notation consists of introducing a system of definitions of variables, and thereafter redefining variables or inspecting their current values. Such a style of programming is particularly well-suited to capturing the semantics of a user-computer dialogue [1]. In that context, the current system of variable definitions is referred to as "the state of the dialogue", and the computational step involved in the redefinition of a variable as "a dialogue action". Allowing the user to program autonomously using variable definitions to capture the current dialogue state is the simplest mode of definitive programming, and requires nothing more sophisticated than a pure definitive notation. In this paper, the same terminology will be adopted for a more general form of definitive programming, in which changing the dialogue state is not the prerogative of the user alone.

The basic technical problems in designing definitive notations, such as the methods to be used when defining variables of complex types that may represent values in many different abstract ways, have been quite fully researched and described elsewhere [2]. An account of the way in which definitive notations can be used to capture many different kinds of abstraction, as illustrated by the specific definitive notations for interactive graphics DoNaLD and ARCA, appears in [2]. The motivation behind the current research on definitive programming is to examine how far the principles developed in the study of pure definitive notations can be applied more generally. In part, this involves identifying those aspects of pure definitive notations that have yet to be exploited. For instance, the introduction of an abstractly defined underlying algebra - incorporating axioms governing the operators - will make it possible to use symbolic manipulation techniques for processing and evaluating definitions. Similarly, greater use of traditional functional programming techniques, such as the introduction of more sophisticated data types, and of higher-order functions as operators, will enhance the expressive power of definitions. To an extent, this paper will indicate respects in which such enhancement of a pure definitive notation can support more sophisticated applications (cf §1.2 below), but it will also be concerned with issues that cannot conveniently be dealt with within the pure definitive notation paradigm.

Each programming paradigm has its own characteristic approach to achieving computational abstraction. In functional programming, a program is regarded as being nothing more than the evaluation of an appropriately defined function, and the user is not encouraged to introspect about the computational process involved in this evaluation. In definitive programming, functional relationships between variables are viewed in a very similar manner: when one variable is functionally defined in terms of others, and its value is implicitly altered through a dialogue action, the mechanism used to update values is deemed to be irrelevant. In effect, any computation that might be involved in carrying out a transition between dialogue states is outside the programming model - or more appropriately in the context of this paper, outside the view of the active agent; the

values of variables are at all times assumed to be consistent with their definition. Updating a spreadsheet is the simplest illustration of this principle in action; the user ideally demands an interface that updates displayed values "instantaneously". In the context of the definitive notation for geometric modelling described below, such idealisation is clearly much further removed from realism - it is in general hard to update a display of a parametrised geometric model in real-time! At some level of abstraction, it is nonetheless appropriate to invoke such idealisation, and possibly to treat other aspects of the user-computer interaction in which computation is involved in a similar manner. It is equally necessary to understand how such an abstract perspective can be practically helpful when building a complete system for geometric modelling, and this concern supplies the context for the study described in §'s 2,3 and 4 below.

1.2 A definitive notation for geometric modelling

The use of a definitive notation for the representation of geometric models is central to the definitive programming approach to CAD software. Designing an appropriate notation is a major part of the conceptual design of the user-interface (cf [12] and §4 below). Pursuing the ideas in [2], it is to be hoped that definitive principles can in due course support feature-based description in parallel with geometric description, thereby avoiding the need for feature extraction for process planning (cf [21],p3). To achieve this, it is not enough to adopt a particular representation for geometric objects; it is necessary to develop ways of integrating many different characteristics of geometric models within a single unifying framework. This is a non-trivial task; as observed in [21], surface and solid models have yet to be successfully integrated in conventional CAD systems.

The relevant characteristics of geometric models include information about reference and construction points, labels, skeletal structure, and those geometric operations that are typically used to synthesise complex objects from simple components. The underlying algebra accordingly includes several different sorts of data, both combinatorial and geometric, together with a system of algebraic operators that can be used to establish relationships between these sorts. The variables over the associated definitive notation can then be used to define models of complex geometric objects, perhaps incorporating components specified using different modelling paradigms, and possibly incompletely specified.

The data types used to represent geometric objects have also been designed to capture the distinction between an abstract object, such as "the sphere with centre c and radius r", and a specific sphere, for which explicit values for c and r are known. The use of a definitive notation lends itself to the representation of such distinctions between partially instantiated and explicit objects, but in a manner that puts too much onus on the user to establish and maintain functional relationships. Using techniques resembling those introduced for moding variables in the definitive notation ARCA [2] it is possible to declare variables to represent objects of a generic form. This obviates the need for the user to repeatedly set up appropriately contrived systems of variables to define explicit objects within the same abstract class. In this way, the designer is able to describe an object in an internally consistent fashion without referring to its location in space, or physical dimensions. This solution to the problem of representing objects is similar in spirit to the use of classes in an object-oriented programming framework.

In essence, the underlying algebra incorporates three distinct sorts for describing geometric objects: **complex**, **frame** and **object**. The **complex** is a combinatorial structure - resembling the abstract simplicial complex of polyhedral topology - intended to capture the purely combinatorial ingredients of a geometric object. These include: reference points, abstract scalars needed to specify lengths and angles etc, and incidence information expressing the way that the object is abstractly synthesised from simpler components. To derive a **frame** from a **complex**, it is necessary to specify the dimension of the space in which the complex is to be realised, and to supply specific coordinates and scalar parameters corresponding to the abstract labels of the **complex**. An **object** is generally determined by a list of **frames**, together with a function that takes the parameters of these **frames** as arguments and returns the extent of the object. (More details of these data types are given in the Appendix.)

The motivating idea behind the choice of sorts is that an **object** is to be viewed as the realisation of a combinatorial structure, as represented by an underlying list of **complexes**. For instance, a spline is determined by a wire-frame together with an appropriate set of boundary elements. The wire-frame has two ingredients: a combinatorial structure, consisting of an array of labelled points with incidence relationships between them, and an associated array of coordinates. By specifying how the spline is abstractly defined in terms of the frame and the boundary elements, without regard for their specific coordinates and scalar values, the spline can be specified as an abstract object. That is to say, the abstract description of the spline takes the form of a function that takes as its arguments an appropriately typed list of **frames** required to specify the wire-frame and its boundary and returns "the set of points that comprise the spline". By subsequently supplying parameters for such a function ie specifying a suitable explicit list of **frames**, a spline is derived as an explicit object.

The design of variables for the definitive notation broadly follows the patterns established in ARCA and DoNaLD [2]. In particular, there is a concept of a variable **mode** - as introduced in ARCA - and of a subvariable - as in the DoNaLD **openshape**. (See the Appendix for more details.) There are also some significant unprecedented features. For instance, an "abstract object" is effectively a component of the **object** data type, so that appropriately moded **object** variables can be regarded as representing operators to realise **objects** from **frames**.

Specifying the **object** data type also presents some novel problems. Since there is no reasonable general method of specifying an object explicitly (ie by enumeration of its points), the extent of an object is represented by a criterion for membership ie a function **vector_of real** → **boolean**. In practice, the relationship between such specifications of objects and the algorithms used to display and manipulate objects may be very complicated. For instance, it may not even be obvious how to apply the criterion for membership of an object in general, and there will be problems of numerical approximation to address [10,13]. There are several pertinent issues to be considered. It is anticipated that **objects** will generally be defined in terms of standard abstract objects (such as spheres or polyhedra) that are supported by built-in functions for display and manipulation in a solid-modelling system. There will be an important role for symbolic manipulation and transformation techniques in relating the abstract definition of a geometric object and its image in the display interface as conceived in §3 below. At the same time, the approach proposed here offers some advantages where the traditional problems of evaluation are concerned, since it is based upon the symbolic representation of objects.

§2 Generalising the definitive programming paradigm

This section examines the limitations of pure definitive notations, and indicates the most promising direction for generalisation of the definitive programming paradigm.

2.1 Limitations of the pure definitive notation paradigm

The use of a pure definitive notation for user-computer dialogue puts the primary emphasis upon representing the current state of the interaction by means of a system of definitions. In practice, this proves to be very effective in as much as the user can readily determine the current state of the dialogue at any time, and can predict the effect of any dialogue actions. However, the use of definitions can also be unnatural, since it requires an acyclic system of functional dependencies between variables.

If the criterion for a good representation of the state of an interaction is "predictability of response to dialogue actions", the restriction to acyclic systems of functional dependencies is superficially unnecessary. It is not even necessary to consider complex constraint relationships (cf §3) to appreciate this. For instance, it may be that two variables **x** and **y** are to be assigned the same value in such a way that if the value of one is changed, then so also is the other. This can be done within

the definitive programming framework - for example, by introducing an auxiliary variable **t**, together with the definitions: **x** = **t**, **y** = **t** - but only in such a way that an attempt to redefine **x** or **y** is construed as a redefinition of **t**. This solution can seem particularly uncomfortable in the context of interactive graphics, where it is congenial to use analogue input via the graphics interface, and direct reference to the internal definitions of variables may not be appropriate. These are the considerations that suggest that a pure definitive program is best viewed as a form of intermediate code for interaction [2].

Appropriate definition of variables can sometimes provide an alternative to hidden parametrisation, but leads in general to a form of over-specification that is somewhat inelegant, and puts an obligation upon the user to recall - or, to be more precise - to refer to knowledge of the representation. For instance, in DoNaLD, it is an easy matter to represent a unit square by defining **point** variables **a**,**b**,**c**,**d** to represent its four vertices subject to the relations:

b = **a**+[0,1]; **c** = **a**+[1,1]; **d** = **a**+[1,0].

To translate the square to another location whilst preserving its orientation, it suffices to redefine the variable **a**, but this is information that is hidden if the user sees only the display.

Such use of functional relationships contrasts with the use of equational relationships between variables commonly employed in a constraint-based graphics system. There are many issues to be examined here (see §3 below): the kind of enhancement of pure definitive notations that can support constraint-processing; the extent to which constraints can be accommodated within the definitive programming paradigm; the significance of representing a constraint using functional relationships.

The passivity of the pure definitive notation paradigm has other important ramifications. In practice, the implementation of an interactive system involves many dynamic elements required for the animation of the dialogue, such as windows, menus and graphical displays. The definitive notation for geometric modelling described in §1.2 can capture the stages of the abstract geometric design process effectively, but does not address the more immediate, if perhaps more ephemeral issues, concerning the current "state of the interaction" in its broad sense. These include considerations such as what windows are currently open, what menu options are currently available, and the form of the responses to input.

There are two aspects to this problem. On the one hand - as is argued in §'s 3 and 4 below - there is good reason to suppose that a system of definitions can be used to represent the entire suspended state of the interface during an interaction effectively. On the other hand, the transitions between dialogue states that are required to model the interface in this manner are very complex, and are in general to be carried out by the computer rather than by the user. To solve this problem, it is first necessary to re-examine the principle of "programming with definitions" to determine how several agents can be allowed to participate in a dialogue. A fuller discussion of the specific implications for user-interface management within a definitive programming paradigm appears in §4 below.

By implication, the generalisation of definitive programming envisaged differs very radically from the elementary use of pure definitive notations. Indeed, to handle the issues of user-interface management within a coherent framework of evolving systems of definitions requires a powerful and general programming paradigm far removed from the special purpose "definitive notations for interaction" described in [2]. An important concern in this paper is to motivate the introduction of a general purpose definitive programming paradigm - after all, other idioms, such as functional or object-oriented programming, have also been advocated for general purpose programming, and provide techniques particularly well-suited for operator specification and data abstraction. This paper argues that the distinctive feature of the definitive programming approach is effective support for the representation of interaction, whether internal or external to a system.

2.2 Enhancing the pure definitive notation paradigm

The method of enhancing pure definitive notations proposed here is based upon abstraction from a mixed programming paradigm first introduced in the EDEN interpreter - a software tool designed as

"an evaluator for definitive notations" [6]. The EDEN interpreter has built-in support for a definitive notation based upon list processing, but can also be programmed to perform traditional procedural actions that may be synchronised with changes in the dialogue state using triggering mechanisms resembling those used in object-oriented systems [8]. By translating definitions into the internal definitive notation, it is easy to represent the state of dialogue over any definitive notation. By using triggered actions, it is easy to make responses contingent upon the current state of the dialogue eg to ensure that a particular screen location in a spreadsheet is to be updated whenever the corresponding variable value is changed, or to register the presence of error conditions such as a constraint violation. EDEN has already been successfully used for the rapid prototyping of software based on definitive principles, including a prototype DoNaLD implementation [6], and is the practical programming tool that is currently being developed to support more ambitious CAD implementation.

In effect, EDEN makes it possible to link complex procedural actions and intricate systems of definitions: a very powerful mixed programming paradigm, but one that can also prove difficult to use and analyse. There is clearly a need to guarantee that triggered actions do not interfere, for instance, and to avoid infinite behaviour. For this reason, triggering has primarily been used to implement actions that have no effect upon the current state of the dialogue as represented by the internal system of variable definitions. Such limited use does not adequately support the direct participation of the computer in the user-computer dialogue that is required for general constraint and user-interface management however. The general definitive programming paradigm to be introduced to this end (cf §'s 3 and 4 below) can be most succintly explained in terms of an abstract machine model that can realise the capabilities of the EDEN interpreter without compromising clarity.

In outline, this abstract machine model consists of three components: a program store P, comprising a set of *entities*, a store D of variable *definitions*, and a store A of *actions*. Each entity comprises an abstractly specified block of definitions and actions, perhaps parametrised, that is superficially analogous to the declaration of a procedure in a conventional procedural language, or of an object in an object-oriented language. The variables whose definitions appear in D may be assumed to be of some basic data type, such as **integer**s or **list**s. At all times, the system of functional dependencies between the variables in D is acyclic, and the value of each variable is consistent with its definition. An action takes the form of a guarded sequence of instructions, each of which either redefines a variable, or invokes the introduction or deletion of a block of new definitions and actions into the stores D and A through the instantiation or elimination of an entity.

A computation consists of a sequence of parallel executions of appropriate actions. In a single computational step, the guards of all actions in the action store are evaluated, and the actions associated with true guards executed in parallel. This in general has the effect of changing the contents of the stores D and A by modifying the definitions of variables (possibly even those whose value is implictly defined by a formula), and may also lead to the introduction or deletion of blocks of definitions and actions. To admit redefinitions involving the evaluation of implicitly defined variables (as is appropriate for instance when referring to the current value of an attribute of an implicitly defined object), there is a mechanism for the evaluation of specified expressions in the same context in which the evaluation of guards is carried out.

In interpreting a user-computer dialogue within the framework of the abstract machine model, the roles of the user and of the computer are determined by entities that are instantiated according to the current context (cf §5 below). There will at any time be variables instantiated to represent new user-input via the keyboard, and to record the present position and status of the mouse, for instance. The abstract machine model then provides a framework that retains the characteristic feature of definitive programming viz the representation of the current state of the user-computer interaction by means of a system of variable definitions, but allows both the user and the computer to initiate dialogue actions to change this state.

Clarifying the abstract machine model is the first step towards the long-term objective of developing

CAD support systems that do not rely upon a semantically complicated mixed programming paradigm. The implications of using the "extended definitive programming idiom" for constraint and user-interface management in CAD applications are examined in more detail in §'s 3 and 4 below, but some of the motivating ideas behind the design of the abstract machine model above may be helpful. *Definitions* are the counterparts at a low-level of abstraction of the high-level definitions that establish functional relationships between variables that represent - for example - complexes of labels, geometric objects and screen displays (cf §1.2 and §4). *Actions* enable autonomous response on the part of the computer, as - for example - when undoing a user-dialogue action that leads to the violation of an imposed constraint, or when automatically invoking a new dialogue context for the user. *Entities* are introduced to meet the need for systems of variable definitions and associated actions that are to be temporarily instantiated as a unit. When declaring an object, an associated family of definitions to describe the associated screen display is required (cf §4), together with actions that are required eg to ensure that specified constraints upon components of the object (cf §3) are imposed.

§3 Constraints and constraint management in the definitive programming model

Constraints play a very important role in computer-aided design. The designer often needs to work in a context where a large number of interdependent constraints have to be met. The functional relationships between variables established using a definitive notation can be very helpful in imposing particular constraints, but there is a need both for additional techniques and for a better understanding of how such functional relationships are connected with general constraints. The purpose of this section is to examine how far constraints and techniques for constraint management can be accommodated within the abstract definitive programming model.

In the first instance, it will be convenient to consider to what extent constraints and systems of variables can be directly integrated, and to explore some elementary techniques that can be used to introduce constraints into the definitive programming model. In general, a dialogue action may have the effect of changing variable values in such a way as to violate a constraint. Even in the pure definitive notation paradigm, such violations can be monitored by using by introducing string variables to flag error conditions. Thus, if C is a constraint condition, then

E = **if** C **then** NULL **else** "C is violated"

defines an appropriate variable E to represent the error status. By displaying E in an appropriate fashion eg in a pre-determined field of an error monitor window, it is possible to carry out a rudimentary form of automatic constraint monitoring. In effect, the user is able to observe the consequences of adverse design decisions in so far as these can be represented using chains of dependent variables.

In the extended definitive programming framework, in which the computer can act to change the state of the dialogue, it is possible to develop this idea further, and provide constraints, that cannot be violated by a user action. To achieve this it is only necessary to set up a protocol whereby a user dialogue action leading to the violation of a specified constraint is automatically revoked. A yet more ambitious objective is the management of constraints in the spirit of a traditional constraint-based system: if the user performs an action that violates a specified constraint, then a predetermined sequence of redefinitions is initiated automatically until the constraint has been restored.

These techniques will be referred to as **monitoring**, **imposing** and **maintaining** a constraint. Both imposing and maintaining constraints require a user protocol that restricts the user's capability for action whilst the constraint is violated. Maintaining constraints of course additionally requires domain specific knowledge, but can be based around techniques that have been well-studied in other contexts [8], suitably adapted. As pointed out in [8], constraint maintenance calls both for declarative information ("what constraint must be met") and procedural information ("how is it to be maintained"). Though it might superficially appear appropriate to view the functional relationships established by definitions as a declarative formulation of a constraint, it will emerge later that they more fruitfully be regarded as concise ways of specifying the methods by which

constraints are maintained.

The critical reader will recognise that the above discussion fails to make a clear semantic distinction between general constraints and "functional relationships established through variable definitions". This is a naive perspective. As illustrated in §2 above, the specification of functional relationships that establish a particular constraint may involve an artificial choice of parametrisation that is aesthetically disturbing. What is more, constraints typically entail more complex relationships between variable values than can be conveniently expressed using definitions alone. As explained below, it is probably much more reasonable to regard a system of definitions as encapsulating one particular agent's view of how a given constraint is maintained (cf the methods considered in [8] §3.2). The techniques described above then become part of a more sophisticated framework for constraint management, in which there need be less emphasis upon constraint maintenance.

To illustrate the latter problem, consider a set of variables

point x, y, z, o; real a, b, c, r

constrained in such a way that **o** is the centre of the circle of radius **r** passing through the points **x**, **y**, **z** with polar coordinates (**r**, **a**), (**r**, **b**) and (**r**, **c**) respectively (see Fig 1). In a constraint-based graphics system, these constraints might be established in such a way that moving the point **o** translates the entire Figure parallel to the axes, that modification of **r** causes the circle to expand or contract so as to respect similarity and preserve the orientation of the triangle **xyz**, modification of the angles **a**, **b** and **c** leads to appropriate relocation of **x**, **y** and **z** on the circle, and relocation of **x**, **y** or **z** causes relocation of the circumcentre **o** of the triangle **xyz** and the associated redefinition of **a**, **b**, **c** and **r**. To represent the rich set of functional relationships implicit in these recipes for constraint maintenance is beyond the scope of an acyclic system of variable definitions. To adequately represent all the geometric transformations involved requires - for instance - functional definitions for **o**, **a**, **b**, **c** and **r** in terms of **x**, **y** and **z**, and definitions for the inverse relationship.

To describe such an interactive environment within the extended definitive programming framework requires a more careful appraisal of the significance of "definitions for interaction", and introduces new concepts to be elaborated in §'s 4 and 5 below. To appreciate the need for a new perspective, it must be remembered that the model of interaction required for extended definitive programming has to take account of several process views. The key idea is that a system of variable definitions can profitably be regarded as representing an interpretation (or perhaps even an "intelligent view") of an object, as observed and possibly manipulated by an agent participating in the dialogue. To be more explicit, the system of functional relationships between a set of variables perceived by a particular process encodes:

(a) the current state of an object,
(b) knowledge of how it can be modified,
(c) what the effect of such a modification will be.

For the object in Figure 1, there are several process views, corresponding to the various ways in which the object is modified by different actions on the part of the user. In this sense, the user acts in the role of several processes - as "the user who points at the centre of the circle" or the user who points at the point **x** on the circumference" etc. Within the extended definitive programming paradigm, this role changing on the part of the user is supported by the user-interface management system that interprets the screen position addressed by the user as a choice of context for the subsequent interaction. By pointing to **o**, the user invokes a state of dialogue comprising a system of definitions for **x**, **y** and **z** in terms of **o**, **a**, **b**, **c** and **r**, and is privileged to change the parameter **o**. By pointing to **x**, the user invokes a state of dialogue comprising a system of definitions for **o**, **a**, **b**, **c** and **r** in terms of **x**, **y** and **z**, and is privileged to change the parameter **x**.

The concept of "a definitive dialogue state as a profile of an intelligent agent" will be further examined in §5 below. It should be noted that the simple protocols used for illustration above are untypical of the context changes that might be programmed in general, and could be enhanced by using more sophisticated input mechanisms (cf §4). The most pertinent point to remark is that - despite the impression that naive enhancement of a pure definitive notation to support monitoring, imposition and perhaps even maintenance of constraints may give - acyclic systems of functional

relationships between variables have a semantics quite different from constraints, and convey information that is not expressed simply by using a constraint. As the above example shows, there are many "intelligent views" of Figure 1 that are consistent with the specified constraints, each corresponding to a different ingredient of the protocol for interactive constraint management. Another important consideration is that certain of these views conflict, in that the associated processes will in general interfere if they should execute concurrently. For instance, the order in which processes are invoked in order to change the parameters **r**, **o** and **a** is not significant - they are non-interfering, but it is not possible to move the centre of the circle and displace a point on the circumference concurrently.

The above discussion focusses on constraints as relationships that a designer may need to impose upon a specific object and its attributes. In a constraint-based programming paradigm, the use of constraints can serve many other purposes. Generic constraints might be used to express algebraic identities, for instance, or to define implicit functions. Notice that these can be modelled in other ways within the definitive programming paradigm, and respectively correspond to introducing axioms and additional operators into the underlying algebra.

§4 The user-interface in the definitive programming model

The abstract problem in designing a user-interface is to represent the current context for interaction to the user in such a way that both the status of the entire system and the options that are available to the user to change the state of the system are as easy to determine as possible. The work that has been done on pure definitive notations has focussed on developing such a representation for the conceptual aspects of the user-computer interaction, but there is no reason why the application of definitive principles should be restricted to such concerns. The purpose of this section is to indicate how the use of the extended definitive programming paradigm described above can in principle deal with all the concerns of the user-interface. Naturally, there are many technical issues to be addressed before this objective can be attained, but some of the key ideas will be outlined.

Following [12], an exchange of information between the user and the computer is conceived as having four ingredients: **conceptual** and **functional** elements concerned with its meaning, and **sequencing** and **hardware binding** elements concerned with its form. The conceptual elements of a geometric modelling system might correspond to the algebra underlying the definitive notation described in §1.2 - or perhaps more realistically to user-defined data types and operators over this algebra, and the functional elements to particular families of entities that are instantiated during an interaction implemented using the abstract machine model described in §2.2. To capture the entire functional design in this machine model, it is necessary to apply definitive principles to the specification of the display, so that the entities may include definitions of variables that are to be interpreted as representing the current state of the screen display. For sequencing purposes, the abstract machine model provides a state machine that has a richer structure than an augmented transition network, in that each state is represented by a family of definitions, and the state transitions can be very subtle and complex. The consistent use of a definitive programming paradigm to deal with interaction at the higher levels of abstraction has the unfortunate effect of highlighting the discrepancy between the software and hardware models; for this reason, there is little to say at present about hardware binding, and it can only be conjectured that the principles used in [14] - for example - may prove relevant.

The design of a definitive notation for screen layout is currently under development (an appropriate notation dealing with general textual displays has already been described). Such a notation will serve to address the issues that are conventionally handled using visual element editors, and provide the basis for the specification of entities that perform the role of the subroutines in the Interaction Technique Library [12]. The principal components in the underlying algebra include data types to represent character strings and graphics, boxes within which text strings are to be laid out or graphics displayed, windows comprising lists of boxes, and displays comprising families of windows. In effect, the current state of a screen display is abstractly described by a system of variable definitions, and it becomes possible to deal with issues of presentation and window

management (as is appropriate for instance when implementing the DoNaLD user-interface) without resorting to the inelegantly mixed paradigm of EDEN (cf §2.2). Perhaps the most exciting implication of using definitions to describe both the objects of the application domain and the screen display is that it readily becomes possible to establish relationships between the form of a display and its content. As a trivial example, the dimensions of a box can be defined to match its contents, in such a way - if necessary - as to reformat the display when exceptional output is to be displayed. Similar principles can be applied to provide dynamic functional feedback to the user (cf [12]).

Amongst the applications for the above method of display are generic methods for dealing with constraints that are to be monitored in the sense described in §3. It becomes directly possible to link the contents of particular "monitor" windows to boolean variables indicating the current status of particular constraints through an appropriate definition. The entities that would be required for this purpose would be easy to generate automatically in reponse to a declaration that a particular condition was to be monitored.

In principle, the methods for displaying information described above can also be adapted to cope with graphical input. As a philosophical point, it seems likely that some emphasis on textual rather than graphical input must be retained however (cf [8], p369), since variable definitions will presumably be hard to introduce using analogue techniques.

As the analysis of constraint-processing in §3 shows, even the conceptual aspects of the user-computer interaction are inadequately represented by a naive use of definitive principles. The user does not in general act within the framework of a single system of functional relationships between variables, but may "select the system of functional relationships" that is appropriate for an intended action. Strictly speaking, such a selection can be articulated directly by the user, who simply needs to preface an action (such as "redefining the parameter **o**" in Figure 1) by an appropriate series of variable redefinitions (such as "redefining the variables **x**, **y** and **z** in terms of **o**, **a**, **b** and **c**"). Though this illustrates how a pure definitive notation supplies a form of intermediate code for interaction, it is clearly inconvenient for the user. Within the extended definitive programming model, subject to introducing auxiliary variables to establish a protocol, the required transition between dialogue states can be handled automatically. To achieve this, the concept of the current state of the dialogue must be enhanced to include information about the user's present intentions, as represented by the definition of a control parameter (such as "the currently selected element" in Figure 1). As an aside, it is of interest to note the resemblance between the user's invocation of an appropriate context for an action, and the generally undesirable use of modes (cf [12] p8). It remains to be seen to what extent the transparent nature of the interface presented by a system of variable definitions alleviates this problem.

In principle, it is clear that dialogue sequence specification can be represented within the extended definitive programming paradigm. A system of variable definitions is a more powerful way to represent state information than an augmented transition network, and - as the above discussion indicates - there is considerable scope for introducing complex transitions between such systems. At this stage, there is much research still to be done into the most appropriate programming techniques for handling control in the abstract machine model of §2.2, but there are several promising indications. It should be easy to accommodate global commands for instance, to support undo actions, and to provide a context sensitivity that encompasses many orthogonal concerns.

Naturally, the use of definitive principles to represent user profiles is advocated as the appropriate approach to developing adaptive interfaces. To achieve this, it will be necessary to establish an "algebraic framework" within which user responses can be monitored and dynamically represented by an evolving system of variable definitions. Such definitions might take the form for instance of numerical data on the proportion of errors made, or commands used, together with boolean variables indicating skills successfully acquired. Notice in particular that the use of definitions avoids issues concerned with the order in which goals are accomplished. The appropriate dialogue context for computer responses can then be determined according to the current status of the user profile. Such a mechanism naturally complements the user's selection of a particular dialogue

context as described above, and requires no greater technical virtuosity.

§5 Definitive programming for intelligent CAD

It can be argued that the representation of state by systems of variable definitions captures an important constituent of human intelligence. When contrasted with procedural or functional models, definitive models appear to abstract relationships that match human conceptual processes more faithfully. Mechanical systems provide strong evidence for this point of view. As a simple example, it is much more appropriate to model an object such as a door as a parametrised system of variables representing the hinge, the handle and the catch than to conceive it as determined by a family of procedural variables that are independently updated in such a way the certain constraints are necessarily satisfied, or as a mathematical abstraction such as a functional programmer might employ that incorporates no concept of current state.

The naivety of proposing such pure definitive models for general systems becomes clear when more complex interactions between objects are involved. As illustrated in §3, systems subject to complex constraints can only be accurately modelled by implicitly introducing many different functional relationships between variables, not all of which can be consistent (ie acyclic). To accommodate this within a definitive framework, it becomes necessary to bind systems to the agents who can observe and act upon them consistently. Rather than modelling "the state of the system", it is then appropriate to model the views of a system as perceived by particular agents. From this perspective, a system of definitions is a way of modelling the view of one agent, who has (conditional) control over certain explicitly defined parameters, and can predict the effect of changing these parameters upon the dependent variables that are implicitly defined. It is in this sense that the definitive programming model of a door is a natural one; it simply expresses the way in which the door can be expected to interact with an agent.

It is of interest to examine more closely the idea - perhaps fanciful - that the concept of "intelligent views" can be an ingredient in a formal framework for studying "intelligence". Guided by analogy with the notion of "intelligent view", it will be appropriate to suppose that our perception of a system can be modelled by a family of variables, and that it is possible to observe the behaviour of the system through changes in the values of these variables with time. In such a context, intelligence about the system might be construed as "knowledge of universal relationships between system variables". There then appears to be a significant distinction between the kind of intelligence that is involved in the perception of constraints, and that involved in "intelligent views" as described above. In effect, there is the intelligence of an observer of the system, who perceives certain invariant constraints between variables (eg as in a system of collinear points A, B and C that autonomously changes so that at all times the distance between A and B exceeds that between B and C), and that of an agent, whose intelligence consists in knowing the effect of actions upon the system (eg if I increase the distance AB by d units the distance BC will increase by the same amount).

The key to unifying the intelligence of the observer and that of the agent appears to be to consider more restricted system models, and to confine the system changes that occur to those that can be brought about by participating agents. It is then no longer possible to speak abstractly of "changes of values of variables in the system with time", but only of actions performed upon the system by legitimate agents. The intelligence of the observer takes the form of consequential knowledge of global constraints upon the system behaviour.

The extended definitive programming model described in this paper seems to offer a most appropriate way to give formal expression to these ideas. A system will be represented by a set of variables that are perceived by various agents to satisfy functional relationships expressed in the form of an acyclic system of variable definitions. The system will evolve from state to state through changes made to parameters - subject to appropriate pre-conditions being met - by the participlating agents. The possibility of concurrent action of two or more agents is not discounted, but there will in general be a need to constrain the behaviour of agents to ensure non-interference, ruling out

concurrent action by two agents that would lead to inconsistent changes in the value of a variable. The global constraints on the system are the invariant relationships between variables - those that cannot be violated by the action of any agents. For convenience, such a system model will be termed a **definitive model**; it may be regarded as describing one possible intelligent interpretation of the system. The nature of the abstraction that is being made in definitive programming (cf §1) also becomes clearer in this perspective; it is not strictly necessary that the maintenance of functional relationships within an agent's intelligent view should be instantaneously updated, but only that these relationships can be guaranteed to persist as postconditions of any legitimate action on the part of the agent, and cannot be subject to interference through the concurrent action of another agent.

There are strong analogies to be made here with the scientific method. "Opening the door" is an experiment that confirms the thesis that the relationship between the essential parameters of the door is correctly perceived. The "intelligent view" of the agent formally represents an assumption that is properly seen as an article of faith, confirmed by experience but unproveable; it may be the case that one day the system will confound the agent's expectation, as when the door comes off its hinges. The quantum theoretic principle that every observer is necessarily an agent [7] also comes to mind.

To illustrate and elaborate the above ideas, it will be helpful to examine a simple example. To this end, suppose that the variables **x**, **y**, **z**, **o**, **r**, **a**, **b** and **c** in Figure 1 supply the basis for a system model. In §2, one possible definitive model consistent with Figure 1 was described. Within that model, the "intelligent views" of the agents were the systems of functional relationships between variables corresponding to the different expectations of the user on selecting different parameters for change. In that model it was coincidentally the case that the variables **x**, **y**, **z**, **o**, **r**, **a**, **b** and **c** were always subject to the constraining relationship graphically depicted in Figure 1.

Of course, there are all kinds of alternative definitive models that might be used to interpret Figure 1. It might be the case that the points and scalar parameters in Figure 1 happened to have the values depicted: in the view of any agent, all the variables would be explicitly defined, and there would be no functional relationships between the variables. This is very much the kind of mental model that underlies conventional procedural programming, and that the discipline of introducing invariants - or object-oriented programming methods - adapts for use in other contexts where there are constraints between variable values to be met.

There are also a number of interesting mechanical interpretations of Figure 1 that might be considered. It may be that **ox**, **oy** and **oz** are the spokes of a wheel; that they indicate the positions of the hour, minute and second hands of a clock; that **ox** and **oy** are diametrically opposite points on a wheel, and **oz** is the point of contact with the ground. To each of these there corresponds a definitive model that describes the effect of agents lifting or rotating the wheel, moving or resetting the clock.

Characteristic of the original model of Figure 1 (as discussed in detail in §2) is the richness of the functional relationships invoked by agents; something that is in general difficult to realise in a mechanical model. This probably accounts for the particular difficulties of programming a user-interface for interactive graphics and CAD software. There can nonetheless be a role in a mechanical system for "switching between different intelligent views" through a simple action similar to the selection of a point on the screen. Suppose for instance, that Figure 1 represents a wheel in which the spokes are bolted together but detached from the rim. Unbolting the spokes has the effect of radically changing the context for an agent who is privileged to be able to move a spoke. What is more, if there were to be an agent for each spoke, the effect would be to eliminate the interference between these agents. It is in the representation of such "intelligent" interactions that definitive principles offer most promise.

The approach to knowledge representation proposed above may be contrasted with the predominantly inference-based techniques commonly used in AI (cf [17]). This relationship has yet to be explored, but promises to be useful both for evaluating and extending the above ideas.

Concluding remarks

The work described in this paper has been motivated by three distinct concerns:

1) a pragmatic concern for developing and implementing sophisticated definitive notations as the basis for CAD software. This aspect of the work is for the present primarily concerned with finding better ways to describe the programming methods that have been used with considerable success for a prototype implementation of the DoNaLD notation (cf [6]).

2) a mathematical interest in developing an abstract programming paradigm that incorporates the principles used in pure definitive notations, but can be applied to more general problems. The abstract machine model described in §2.2 is what appears to be the most appropriate generalisation, in the light of experience with related work on EDEN [6] and LSD [5].

3) a semantic concern with fundamental principles of interaction, and the significance of definitive principles as a way of representing knowledge about systems in a psychologically convincing way. Where pure definitive notations are concerned, some of the most important issues have been already addressed (cf [2]), but the perspective presented in this paper suggests further potential within a less restricted programming model.

Respectively associated with these concerns, there are three broad objectives: implementing large software systems that exploit definitive principles, describing a satisfactory semantics for definitive programming, and developing new applications.

The implications of these three strands of research are at present only partially understood. The machine model described in §2.2 has as yet been little developed, for instance, and considerably more research is required before the methods currently being used for implementation can be recast in an abstract form. Work is in progress on the development of the definitive notation for geometric modelling (cf §1.2 and the Appendix), and on a definitive notation for describing presentation issues in the display interface (cf §4). The problems posed by concurrency, and the programming techniques needed to deal with synchronisation issues are also a particular focus of current concern. What is now clear is that there is no satisfactory way to compromise in the use of definitive principles; whatever the shortcomings of the extended definitive programming framework outlined in this paper, there is a strong motivation to fulfil the objectives set out above. As a footnote, it may be worth observing that the computational model that is perhaps most appropriate at the primitive hardware level, where the output of a gate is directly determined as a function of its inputs, closely resembles a definitive program!

The extended definitive programming framework described in this paper has something in common with both functional and object-oriented programming methods. The specification and augmentation of the data types and operators of the underlying algebra offers much scope for functional programming techniques such as are to be found for instance in [24]. There are clear connections between the methods for constraint processing and user-interface management discussed in §'s 3 and 4 and the application of object-oriented principles as described in [8]. Characteristic of the definitive programming approach is the explicit identification of relationships between variables as viewed by an agent; a significant form of abstraction that cannot be made explicit in object-oriented models but is often implicit in an object-oriented implementation. The use of trigger constraints by which "responses are keyed to particular message selectors to be received by specific *instances* " (identified as a powerful feature of the BALSA animation system in [8] p369) is one example of an implementation technique that provides convenient support for such relationships. It is of particular interest to see that variants of definitive programming have been implemented in many different programming paradigms: cf the data-base language ISBL [23] (in a PL1 environment), The Analytic Spreadsheet [11] (in an object-oriented environment), and the geometric design tool RELATOR [19] (in a Prolog environment). This may be construed as further evidence that the use of variable definitions to represent relationships is of significant interest, but outside the immediate scope of traditional paradigms.

Acknowledgements

The research described in this paper owes much to the support of Samia Meziani, Mike Slade and Edward Yung. We are also indebted to the University of Warwick Research and Innovations Fund, and to the Computer Science Department, for financial support.

References

1: W M Beynon, *Definitive notations for interaction*, Proc hci'85, CUP 1985, 23-34
2: W M Beynon, *Definitive principles for interactive graphics*, NATO ASI Series F; Computer and Systems Sciences, Vol 40, 1083-1097
3: W M Beynon, *ARCA: a notation for displaying an manipulating combinatorial diagrams*, Univ of Warwick RR#78, 1986
4: W M Beynon, D Angier, T Bissell, S Hunt, *DoNaLD: a line drawing system based on definitive principles*, Univ of Warwick RR#86, 1986
5: W M Beynon, *The LSD notation for communicating systems*, Univ of Warwick RR#87, 1986
6: W M Beynon, E Yung, *Implementing a definitive notation for interactive graphics*, Proc CG'88
7: Niels Bohr, *Atomic Physics and Human Knowledge*, Science Editions Inc, NY 1961
8: A Borning and R Duisberg, *Constraint-Based Tools for Building User Interfaces*, ACM Transactions on Graphics, Vol 5, No 4, October 1986, 345-374
9: U Cugini, *The role of different levels of modelling in CAD systems*, NATO ASI Series F; Computer and Systems Sciences, Vol 40, 881-898
10: R T Farouki and J K Hinds, *A Hierarchy of Geometric Forms*, IEEE Computer Graphics & Applications, May 1985, 51-78
11: J J Florentin, *The Analytic Spreadsheet of Objects*, OOPS 11: Beyond the Beginning, 1987
12: J Foley, *Models and Tools for the Designers of User-Computer Interfaces*, NATO ASI Series F; Computer and Systems Sciences, Vol 40, 1121-1152
13: R Forrest, *Geometric Computing Environments: Some Tentative Thoughts*, NATO ASI Series F; Computer and Systems Sciences, Vol 40, 185-198
14: P ten Hagen, R van Liere, *A model for graphical interaction*, NATO ASI Series F; Computer and Systems Sciences, Vol 40, 517-542
15: K E Iverson, *A Programming Language*, Wiley 1962
16: M L Kersten, F H Schippers, *Towards an object-centered database language*, Report CS-R8630, CWI Amsterdam 1986
17: J Lansdown, Graphics, *Design and Artificial Intelligence*, NATO ASI Series F; Computer and Systems Sciences, Vol 40, 1153-1174
18: G Nelson, *Juno, a constraint-based graphics system*, SIGGRAPH '85, 235-243
19: M Y Rafiq & I A MacLeod, *Logic Programming for Technical Design*, Workshop on AI in Civil Engineering, AI Applications Institute, Edinburgh University, November 1987.
20: R F Riesenfeld et al, *Computer aided design*, UUCS-84-003, CS Dept, Univ of Utah 1984
21: T Smithers, *AI-Based Design v Geometry-Based Design*, Workshop on AI in Civil Engineering, AI Applications Institute, Edinburgh University, November 1987.
22: T Takala, C D Woodward, *Industrial design based on geometric intentions*, NATO ASI Series F; Computer and Systems Sciences, Vol 40, 953-964
23: S J P Todd, *The Peterlee Relational Test Vehicle - a system overview*, IBM Syst J 15, 285-308
24: D Turner, *An Overview of Miranda*, Bull EATCS, Oct 1987, 103-114
25: C J Van Wyk, *IDEAL User's Manual*, Computing Science TR #103, Bell Labs, 1981
26: K Weiler, *Edge-based data structures for solid modelling in curved surface environments*, IEEE Computer Graphics and Applications, 1986, 21-40

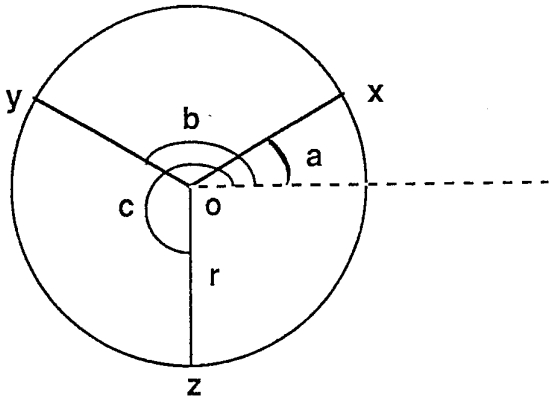

Figure 1: A system of constraints

APPENDIX: A definitive notation for geometric modelling

In this appendix, we give more details of the definitive (definition-based) notation for the description and manipulation of geometric objects in 3-dimensional space outlined in §1.2.

Background

There have been many different ways to approach the computer representation of complex geometric objects. It may be most natural to represent an object using solid-modelling techniques for instance, expressing it as the result of applying a sequence of set-theoretic operations to a family of standard objects, such as spheres and cuboids. Alternatively, we might wish to synthesise an object from a combinatorial structure, such as a wire frame, using surface modelling techniques for realisation of the complete object. To an extent, the disparate views of geometric structure may be seen as the consequence of a natural development in modes of computer use, reflecting the historical progress of computer-aided design, and the nature of the computing resources available [9]. The usual approach to modelling has been to adopt a single consistent paradigm for the description of objects, according to the nature of the geometric model required. Developing the ideas proposed in [2], we should like to construct an underlying algebra for our definitive notation that enables us to make many "orthogonal abstractions" from a geometric object, and aim to integrate several different kinds of geometric model within a single framework.

A simple illustrative example may be helpful at this point. Suppose that our task is to model a simple table lamp, comprising a base consisting of a spherical section, attached to a rectangular block supporting a hinged jointed arm, linked by a flange to a parabolic reflector. It might be most appropriate to model the base as the set-theoretic intersection of a sphere and a half-plane, and to incorporate the rectangular block by forming a set-theoretic union. To represent the jointed arm of the lamp in such a way that its different possible positions could be conveniently depicted, and in such a way that the implications of choosing arm segments of differing lengths could be examined, it would perhaps be simplest to synthesise the arm from a combinatorial description giving parametrised positions for the hinges at the base, the midpoint and the flange. The flange might be best described by smooth surfaces realised from an appropriate system of construction points within a wire frame model, and the reflector as the surface of rotation obtained from an appropriate parabolic arc.

Within our underlying algebra, there will be elements of several different sorts. A variable of type **object** might represent a geometric object as the result of applying an algebraic operator to realise a surface from a **frame** (a combinatorial complex resembling a wire frame model), and the incidence structure of such a **frame** might in turn be abstractly defined as the result of combining elementary components. The complete lamp might be represented by a variable L of type **object** defined as the union of 6 variables of type **object** respectively representing the reflector, the flange, the two jointed segments of the arm, the rectangular block and the base. The reflector (for instance) might then be represented by a variable R defined by an algebraic expression identifying it as the locus of rotation of a planar curve C about an appropriate axis A, where C and A are again variables of type **object**, and C is defined as a quadratic spline based upon a system of construction points specified using a variable of type **frame**. The explicit set of points associated with the **object** variable R would depend upon the position of the plane defined by the curve C and by the point of intersection between C and A, which in turn would be defined relative to the flange. The position of the flange F would be determined with reference to the location of the uppermost hinge of the arm, and the position of the arm itself specified in terms of free parameters indicating the lengths of its jointed segments, and the angles of aperture at each of its joints.

The development of a complete representation for the table lamp in such terms is left to the reader's imagination. It should be clear that such a representation can be defined in such a way that relocation of the lamp, as in rotating and translating the base, leads directly to the relocation of all the constituent parts. What is more, we are free to mix geometric components defined using quite different modelling paradigms, and to build in complex systems of parametrisation.

An overview of the definitive notation

A definitive notation is specified by first identifying an underlying algebra of sorts and operators. For our purposes, this must be a rich algebra, making it possible to describe many different structural aspects of a geometric object. The complete specification of the syntax and semantics for the operators in such an algebra is a very formidable task, and our emphasis will be upon providing the conceptual framework for the sorts and operators we shall require, rather than upon a detailed and comprehensive description. This is in no way to evade the main issues; the formal description of operators that are simply pure partial functions between various sorts is a routine exercise once an appropriate classification of sorts and operators has been established. (The difficulties are non-trivial only in so far as it matters to the user how these operators are syntactically represented: Iverson's remarkable syntax for linear operators in APL [15] illustrates how significant this representation can be.)

I. Data types in the underlying algebra

In developing the underlying algebra for our definitive notation, we shall distinguish three essentially different ways in which a geometric object can be viewed: these will be respectively associated with the sorts **complex**, **frame** and **object**. In describing these, we shall assume that our algebra incorporates some primitive auxiliary types and operators to support lists, character strings, scalar and vector variables.

A **complex** comprises a list of labels (character string identifiers), together with a list of subsets of this set of labels. The labels are to be viewed as references to abstract points that lie in an Euclidean space of dimension $d \geq 1$; this means in particular that labels may refer simply to abstract scalars. The non-negative integer d is then the dimension of the **complex**, but is not specified as part of the data type. Our notion of **complex** corresponds closely to the "abstract simplicial complex" of polyhedral topology, but we shall not wish to confine the interpretations of our simplices to objects that are topologically equivalent to simplices. The intention is rather to use complexes to capture the purely combinatorial ingredients of a geometric object. Such ingredients include: reference points and dimensions, and incidence information expressing the way that the object is abstractly synthesised from simpler components. In particular, the reference points that appear in the underlying **complex** of a geometric object will be significant when accessing an object through the graphics interface. Note that there is no coordinate information in a **complex**: physical locations of points are not represented at this level of abstraction.

To give more geometric substance to a **complex**, we may specify its dimension, thereby obtaining a **d-complex**. The reason for introducing these two very similar data types is that combinatorial operators are most naturally specified on **complexes**, but the specification of **frames** and **objects** typically requires knowledge of the dimension of the underlying **complexes**. A **frame** consists of a **d-complex** together with a list of coordinate vectors of the appropriate dimension whose role is to supply locations for the vertices. Even in this form, a **frame** remains an abstraction from an **object**; though there is a canonical way to realise a **frame** as an **object**, the intention is that a **frame** supplies a finite set of parameters that can be used to represent the essential reference and construction points from which an **object** is synthesised. It will be convenient to regard the **d-complex** associated with a **frame** as supplying a type for the **frame**; in general the operators that are specified on **frames** will act only upon **frames** of specified types.

The most sophisticated of the basic data types is the **object**. Our idea is to represent an **object** abstractly, not as specified by its extent (ie the set of points deemed to be within the object) alone, but by the ingredients from which this extent is in general determined, viz: the combinatorial information needed to obtain an abstract criterion for membership of the object, together with the specific values for relevant vector and scalar parameters. As a simple example, a solid sphere can be regarded as determined by two parameters: its centre **c** and its radius **r**. The abstract criterion for membership of the solid sphere can be expressed as "$|\mathbf{x}-\mathbf{c}| \leq \mathbf{r}$"; assigning specific values to **c** and **r**

will determine a specific solid sphere.

The above discussion motivates the following specifications for our basic data types:

complex = **list_of label** × **list_of list_of label**
- where each list of labels in the second component is a sublist of the list of labels specified in the first component.

d-complex = **complex** × **int**
- where the integer parameter supplies the dimension of the Euclidean space in which the specified complex is to be realised.

frame = **list_of vector_of real** × **d-complex**
- where the vectors associate coordinates with the labels of the **d-complex**, and all the points within a **frame** are in a Euclidean space of the specified dimension. (The **d-complex** then supplies the type of the **frame**.)

object = **list_of frame** × (**list_of frame** → (**vector_of real** → **boolean**))
- where the first argument supplies appropriately typed parameters for the function defined on the list of **frames**, and both lists may be empty.

As explained above, it is useful to be able to distinguish between *abstract* objects that are formally specified in terms of an underlying combinatorial structure, and *explicit* objects that correspond to a particular set of points in an Euclidean space. An appropriate data type to represent an abstract object might be:

(abstract) **object** = **list_of frame** → (**vector_of real** → **boolean**)

- though the methods of variable moding and definition that we introduce make it possible to describe generic objects of a wide variety of kinds, of which such an abstract **object** is just one. Our definition of objects directly supports a mapping from explicit to abstract **objects** in an obvious way, making it possible to associate an abstractly defined object with each fully instantiated object.

A **d-complex** can be viewed as a generic **frame** of the appropriate type. In effect, there is an implicit function mapping **frames** to **d-complexes** that forgets the coordinate structure. When specifying an abstract **object**, it is convenient to think of "writing a boolean condition for a point to be within the object in terms of the abstract points referenced by the labels in a list of **d-complexes**". For instance, the sphere with centre **c** and radius **r** is defined as an abstract object by the boolean condition $\{x \mid |x\text{-}c| \leq r\}$ parametrised by the pair of **d-complexes** that specify its centre as an abstract point of dimension 3 and its radius as a abstract scalar. The list of **d-complexes** that is required for such specification of an object will be viewed as its type; this is a way of expressing the fact that - for instance - the abstract object "sphere" is a function whose parameters must consist of a centre and a radius.

There are several reasons for requiring a list of frames for specifying objects in general. When specifying a spline, for instance, it is helpful to be able to distinguish between the underlying frame and the set of boundary points. It is also important to distinguish between parameters that supply scalar and coordinate information. An object may also be built up by synthesising components from subcomplexes using a variety of operators eg realising some simplices rectilinearly, and others as circular or spherical components.

It at first seems paradoxical that we provide techniques for declaring variables in such a way that they can represent "parametrised objects". After all, definitive notations are very well-suited to the direct representation of such parametric relationships, and it is certainly possible to achieve the effect of having a parametrised object by giving appropriate definitions to variables of type **object**. The merit of the approach we have adopted is that it becomes possible to specify *generic* parametrisations, as will be illustrated below. (To clarify the issues involved, it may be helpful to

contrast the way in which a wire frame might be represented in a less sophisticated definitive notation such as DoNaLD by explicitly setting up a family of **point** and **line** variables to represent triangles, and defining the vertices of these triangles in such a way that they coincide in the appropriate manner.)

To complete the underlying algebra, we introduce an additional data type. The view we shall take is that an **object** is analogous to a DoNaLD **point** or **line** (indeed these can be viewed as subclasses of **object** in the sense introduced here), and there will be an analogue of the DoNaLD **shape** in the form of a **suite** of **objects**. As will be explained in more detail below, variables of type **suite** will be treated in the same way as DoNaLD variables of type **shape**, whilst variables of type **complex**, **frame** and **object** will be treated in much the same fashion as ARCA variables, and ascribed a *mode* using a technique for incremental declaration.

II. Operators in the underlying algebra

The underlying algebra for our definitive notation is based around the following categories of operators: combinatorial operators that are used to synthesise new complexes from old, geometric operators that assist the specification of coordinates for frames, operators that combine objects as in traditional solid modelling, and operators to specify objects, frames and complexes in terms of each other.

Combinatorial operators on complexes

Since **complexes** are synthesised from **lists** and **strings**, the underlying algebra will include the traditional operators for list and string processing. In all cases, it is intended that operators are specified as pure functions; typical functions served by such operators are: concatenation of lists and strings, extraction of elements from lists using head(), tail() and perhaps more general projection operators, and auxiliary higher order functions such as maplist() that will modify a list by applying a specific operator to each of its elements. Combinatorial operators on complexes can readily be derived from these basic operators, though it may be important to incorporate certain special operators as built-in functions in the interests of efficiency. This reflects the philosophy behind our approach; it is impossible to anticipate the many ways in which the combinatorial structure of objects may be specified or related, and a toolkit of basic operators is supplied.

There are many ways in which it will be useful to combine or modify complexes. Forming the union or intersection of complexes, restricting a complex to a subset of the set of vertex labels, forming the n-dimensional skeleton, or extracting the boundary are typical of the standard operators that can be readily specified formally. The underlying algebra will also include operators of mixed type, such as those that return the abstract dimension of a simplex, the number of vertices in a **complex**, and the list of vertex labels. Operators that modify the list of vertex labels by relabelling may also be required.

It is important to note that the operators acting on complexes must require only combinatorial input: ie they should not be specified in terms of any metric or geometric information. For instance, the location of vertices (eg coincidence, collinearity, coplanarity of vertices) should be irrelevant. Operators whose arguments require such geometric information will apply to **frames** rather than **complexes**.

Operators on frames

A **frame** differs from a **complex** in that it associates coordinate information with the vertex labels. The operators associated with frames can be classified according to their effect on sorts:

(1) constructors and selectors to put together and unpick frames into parts;

(2) operators defined within the constituent parts: all the incidence operators for complexes, and appropriate geometric operators for constructing a **vector_of real**;

(3) operators that require as arguments both combinatorial and geometric information and

return a new **frame**;

(4) operators that require as arguments both combinatorial and geometric information and return an **object**.

The operators under (1) and (2) are easy to conceive, and will not be considered in detail here: the combinatorial operators have been described, and the specification of coordinate vectors can broadly follow the patterns established by ARCA and DoNaLD [2,3,4].

Under (3), there are operators such as "form the convex hull" which are meaningful only when the locations of vertices are specified, and whose effect can be interpreted unambiguously in terms of frames (ie as deterministically modifying the incidence structure under consideration). Subdivision of a frame using midpoints, and extraction of visible components also fall into this class. Other operators that might be introduced include generalisations of the operators on **complex**es such as forming the union of two **frame**s, forming the **frame** comprising the boundary of a given **frame**, or returning a **frame** comprising the centre and radius of the circumscribing sphere associated with an embedded simplex.

A full treatment of the operators under (4) properly belongs to the section on operators that return objects below, but these include some standard operators for the realisation of an object from a frame. For instance, the simplices of a frame comprising simplices of dimension ≤ 3 may be directly interpreted as rectilinear objects, or realised as circles or spheres according to their dimension. Splines also provide a generic method of realising appropriate frames as geometric objects. As explained in more detail below, operators to realise objects will commonly require a list of frames as input; for instance, a spline will generally require additional information concerning boundary conditions.

Specifying the dimension of the space in which the coordinates for frames lie is an important issue. It is important to be able to do local geometry within any affine subspace when defining objects: for this reason, it is convenient to have a way of specifying the basis of the vector space within which coordinates are given. To this end, the coordinates of points may be most appropriately specified in homogeneous coordinates, relative to a specified basis vectors. The expression "(a,b,c)@(x,y,z)" would then be interpreted as "ax+by+cz relative to the basis vectors x,y,z" - a point in the plane spanned by triple of points x,y and z. Adopting this technique from classical affine geometry makes it possible to refer to "points at infinity", and offers several computational advantages. These include: convenient use of a basis as a parameter for the specification of coordinates; simple ways of specifying geometric transformations; provision for essentially the same frame to be used as a blueprint for an object in several different subspaces. Note that in this context the explicit values of vectors will be specified in world coordinates.

As will be explained below, providing a basis as a parameter is one of several techniques for parametrising **frame**s that can be exploited when defining **frame** variables to ensure that a **frame** satisfies required constraints. In general, moding of **frame** variables will permit the description of many different generic classes of **frame**.

Operators on objects

There are essentially two types of operators associated with objects: those that can be used to specify objects, and those that permit objects to be manipulated and composed. Within these categories, there is a further distinction between built-in and user-defined operators.

Abstract objects provide simple examples of operators that can be used to specify explicit objects. As explained above, an abstract object can be viewed as a function of the following type:

list_of frame $\rightarrow$ (**vector_of real** $\rightarrow$ **boolean**),

where the list of **frame**s is typed to match an appropriate list of **d-complex**es. By definition, supplying an suitable list of **frame**s as a parameter for such a function creates an **object**, so that in

effect an abstract **object** can be viewed as an operator:

list_of frame → object.

As operators specifying objects, abstract objects may be contrasted with generic operators whose range is not restricted to lists of **frames** of a particular type.

The most primitive generic operator for realising a **frame** as an **object** is the rectilinear realisation of a **frame**. This is an operator of the sort:

frame → object,

that is polymorphic with respect to the type of **frame** supplied as a parameter. That is, given a **frame** it is possible to derive a boolean function, expressed solely in terms of the vertices of the associated **d-complex**, that specifies the conditions that a point must satisfy in order to be within the rectilinear realisation of the **frame**. Other examples of generic operators can also be contrived; for instance, it would be easy to introduce an operator to realise frames comprising simplices of dimension at most 3 by interpreting each constituent simplex as a solid sphere or a circular disk according to its dimension, and perhaps to provide for the realisation of boundaries of such simplices within a complex as spherical surfaces, or circular rings. Splines offer a more complicated example of a class of operators sharing a common form; in this case, each particular type of spline would be associated with a family of operators of the sort:

list_of frame → object,

where the list of **frames** matches a list of **d-complexes** of the appropriate types. Similar considerations apply to objects specified as loci eg as volumes or surfaces of rotation.

The basic operators for constructing objects will in general be used in conjunction with standard operators for synthesising objects. These include: forming the **intersection, union** or **boolean sum** of a family of objects. The specification of these operators is quite straightforward: and will be illustrated in the case of the union operator only. Suppose then that A and A' are **objects**, and

$$A \equiv < [\mathbf{F}], B([C]) > , \quad A' \equiv < [\mathbf{F}'], B'([C']) >$$

where [**F**] denotes a list of **frames** of type consistent with the list of **d-complexes** [C], and B([C]) denotes a boolean criterion for membership of A etc. The union of A and A' would then be

$$U \equiv < [\mathbf{F}] ++ [\mathbf{F}'] , (B \textbf{ or } B')([C] ++ [C']) >$$

where "++" denotes a list concatenation. (Note that the labels within the lists of complexes [C] and [C'] are assumed disjoint: some syntactic ingenuity is needed to ensure this.) The union operator not only serves to form the union of two objects viewed as point sets, but can also be used simply to introduce extra points and components that can be referenced.

III. Variables in the definitive notation

Our definitive notation will include variables to represent geometric objects both of type **frame** and type **object**, together with variables of type **suite** that represent families of **objects** in much the same way that DoNaLD **shapes** represent families of **points** and **lines**.

The declaration of variables of type **suite** follows the conventions established for **shape** variables in DoNaLD: each **suite** comprises **objects** and **subsuites**, and the value of a **suite** variable is either defined via an expression of type **suite**, or defined component by component using appropriate definitions to give values to its constituent **objects** and **subsuites**. The ideas already developed for the DoNaLD **shapes** also provide a suitable environment for handling such **suite** variables; for instance, each **suite** will be associated with a context window that includes the definitions pertaining to the variables within its scope. Since there is no essential novelty in the generalisation from **shape** to **suite**, the reader is referred to [2] and [4] for fuller details.

Where variables of type **frame** and **object** are concerned, different conventions are required. We have already identified the need to specify **objects** both abstractly and as fully instantiated sets of points in Euclidean space. In general, a high degree of control over the nature of the abstraction of

an **object** represented by a particular variable is needed, and this will be supported by a method of variable moding first introduced in the definitive notation ARCA [2]. According to this pattern, the declaration of a variable of type **frame** determines the way that it is used to represent a value of type **frame**. The simplest declaration we can make takes the form:

mode F = **abst frame**;

this indicates that the variable F is to be defined in its entirety by an algebraic expression of type **frame**. If the form of the **frame** to be represented by the variable F is to be specified through its components, an alternative declaration must be used. Developing the ideas introduced in §3, we propose to extend the range of such **mode** declarations to make it possible to specify generic constraints between the components of variables of a given **mode**. This can be done using one of three techniques, through:

1) explicit parametrisation of objects in such a way as to satisfy constraints
2) imposition of constraints so that a dialogue action leading to a violation is revoked
3) monitoring of constraints so that all violations of constraints are registered as and when they pertain to the current dialogue context.

The most general template for such the declaration of a frame variable is then:

mode F = **frame on** <complex>
where <labelled points are appropriately defined>
subject to <specified imposed and/or monitored constraints>.

Semantically: the variable *F* represents a **frame** with labelled vertices and combinatorial components as specified in the <complex>, satisfying the relationships between vertices defined in the in the **where**-clause, subject to constraints whose nature and form is as specified.

Similar considerations apply to the declaration of variables of type **object**, where the **mode** must be in general be specified according to the template:

mode frame_1 = ... ; **mode** frame_2 = ... ; .....; **mode** frame_N = ... ;
mode O = **object on** [
frame_1, frame_2, ..., frame_N
]
where <labelled points are appropriately defined>
subject to <specified imposed and/or monitored constraints>
with extent <parametrised boolean condition>.

In this **mode** declaration, there need be no **where**, **subject to**, or **with extent** clauses. Nor is it necessary to specify the **mode** of the component variable to represent the extent of the **object**. Note also that an **object** variable can be declared so that it represents an abstract **object**.

Outstanding issues and further directions

The aim of this Appendix has been to demonstrate the viability of a definitive notation for geometric modelling that is not restricted to a single modelling paradigm. Our objective has been to address the main conceptual problems that have to be solved; many technical issues remain to be considered. The design of an appropriate syntax is important, and must be guided by the intended use. At this stage, it is not clear how frequently the user will need to specify the extent of an object by introducing an explicit boolean condition, nor how far such specification could be supported algorithmically. For instance, we could introduce an expression to represent the intersection of a sphere, and a surface of rotation, but it might be very complicated to determine whether or not a particular line intersected the corresponding patch of surface (cf [10]). We do not pretend that our approach in itself provides new solutions to the formidable algebraic and numeric problems of geometric design, but believe that it can offer full support to existing techniques. Moreover, our symbolic representation of **objects** potentially has considerable advantages over an explicit representation, and it is to be hoped that appropriate techniques for transformation and symbolic manipulation can be introduced to address the significant issues of computation and display.