Definitive programming as a framework for design

Meurig Beynon
Dept of Computer Science, University of Warwick, Coventry CV4 7AL, UK
Telephone: +44 (203) 523089 Telex: 31406 COVLIB G Fax: +44 (203) 461606
E-mail: wmb@uk.ac.warwick

Introduction

Research on design support systems has traditionally been concerned with two themes. One is the development of tools with powerful problem solving abilities in restricted problem domains. The other is with developing better interfaces between the designer and the system. These two concerns are complementary. Making tools more and more sophisticated and capable of autonomous action is intended to enhance the role played by the computer in the design process. Building more intelligent user interfaces aims to assist the designer in articulating a design. The challenge is to build systems that allow the designer to exploit the computational power of modern computers without becoming locked out of the design loop in the process.

Trends in recent research reflect the need to address this fundamental problem. The IIICAD (Intelligent Integrated Interactive CAD) system, as introduced in [19], aims to provide "an integrated infrastructure for problem solving tools together with a smart user interface". What is involved in building such an integrated infrastructure? Establishing common data representations that allow a simple integration of CAD systems and AI tools is not enough. The system components have to operate in a manner that can be intelligently controlled by the designer. The IIICAD system proposes to solve this problem through integrating the design knowledge associated with the tools, and introducing an intelligent supervisor to control the system on the basis of this knowledge.

This paper approaches the problem of realising the ideals of IIICAD in a different way. The difficulties of integrating powerful computational tools within a system that accommodates the human designer are interpreted as reflecting the inadequacy of current methods for representing computation in an interactive environment. It is argued that by choosing a better programming paradigm for describing interactive computation, it will become possible to make complex computational processes intelligible to the designer to a far greater degree. This will enable the designer to interact more closely and directly with the tools.

In understanding how to apply new principles to interaction within a design support system, it is important to clarify their potential role and scope. There is clearly a place for tools whose computational activity is opaque to the designer. It may be neither appropriate nor practical to equip the designer to intervene in the technical computations performed by tools, e.g. in numerical computation, or routine catalogue searching. But there will also be computational tasks within the system in which the designer has to be centrally involved, and where the raw application of computing power is no substitute for human insight and intuition. Much of the activity associated with the construction and validation of a design object is of this nature. It is in supporting this interaction that an appropriate representation for computation is crucial.

The problem of representing the computation carried out by the system to the designer in a useful manner is fundamentally linked to the choice of the underlying programming paradigm. This paper develops proposals made in previous papers [4,5,6], further exploring a computational model that exploits a definitive (definition-based) programming paradigm. Definitive principles are particularly well-suited to expressing computation in which there is interaction between user and computer. They are intended to give the designer means of directly effecting state transitions in a computation, of setting up autonomous computational processes to transform, simulate or realise a design object, and of suspending and intervening in such computational processes where appropriate.

The four sections of the paper respectively concern the motivation and basic concepts of definitive programming, the proposed application of definitive principles to design support, some simple illustrative examples of their use, and a brief comparison with alternative approaches.

§1. Definitive programming: motivation and basic concepts

What will the design environments of the future look like? It is to be hoped that the designer will be able to build a representation of the design object, to simulate its behaviour, and to customise the design environment itself. The latter involves the interactive modification of the design tools to assist the processes of flexible representation and effective simulation. Present techniques aimed at addressing these issues are diverse and hard to integrate: the representation of design object will perhaps be in form of information stored in a sophisticated data base; its behaviour will be described by means of qualitative reasoning expressed in terms of inference mechanisms; adaptation of the design tools will typically involve editing the specification of a piece of software.

From the perspective of this paper, the evolution of the design object as the design develops, the changing state of the design object as it is transformed in simulation, the modification of the design interface to suit the designer's present needs are all regarded as computations that unfold during the design process. Each is a computation which the designer has to comprehend, and be able to direct accordingly. They are computations in which the concept of a current state is strongly represented, and in which the course of the entire computation cannot be preconceived.

The development of programming paradigms has been heavily influenced by the need to hide computation from the user. Surely the designer does not need know the elaborate computations required in displaying a geometric object, or to understand the procedure by which a particular data item is retrieved from a catalogue. It is sometimes essential for clarity to use declarative abstractions that describe what is being computed without reference to the computational recipe to be used. But it is hardly to be expected that the programming principles that hide computation are also appropriate for interactive computation in which cooperation between the user and the computer is required. This is the principal motivation for a novel programming paradigm based upon the use of definitions: definitive programming. This section provides an overview of definitive programming, and introduces the concepts to be discussed and illustrated in later sections.

Definitive programming is a state-based programming paradigm in which the central abstraction is the *definitive system*. A definitive system is a family of variables such that the value of each variable is either specified explicitly, or is defined by a formula in terms of constants and other variables. The values of the variables in a definitive system can be seen as determining the current state, and a definitive system as representing a possible way in which the current state can be changed. The potential state transitions represented by a definitive system are associated with the redefinition of one or more of the explicitly defined variables.

In interpreting a definitive program, it may be useful to think in terms of a state transition model. Though the current computational state is strictly speaking represented by a set of values, it is often appropriate to represent this via a system of definitions. As explained above, such a system of definitions expresses latent state transitions associated with the redefinition of explicitly defined variables within the system. Following the conventional method of presentation of a state-based machine model, the computational state (as represented by the set of values associated with variables in a definitive system) can be conceived as a node of a graph, and the possible transitions from this state (as represented by possible changes of explicitly defined variables within the definitive system) as edges directed outward from this node. The combinatorial graph specified in this way will be very hard to conceive in all but the very simplest cases, but provides a faithful representation of the rich computational structure associated with a definitive program. Since several independent transitions associated with a single definitive system can be non-interfering, there is also the possibility of composite state transitions that in effect correspond to concurrrent execution of several redefinitions. Such a model of concurrent computation underlies the abstract definitive machine described below.

In the simplest form of definitive programming, the transition from state to state is entirely under the control of the user. The user generally describes an initial computational state by formulating appropriate definitions, and subsequently changes the state by redefining variables. Such a style of programming subsumes the underlying principle of the spreadsheet. A definitive system may be used to define the profit as a function of costs and sales, for instance, so that the expected profit

changes according to the predictions for costs and sales. Definitive systems can be formulated for other applications, when the values are not merely scalar, and the operators not merely arithmetic. The values and the operators that can be used to combine values in a formula constitute an *underlying algebra.* By choosing the underlying algebra appropriately, a wide range of applications can be addressed [5]. It is possible to use a definitive system to describe the screen layout, for instance, so that updating the screen involves redefining appropriate parameters.

The use of a single definitive system over a sufficiently rich underlying algebra itself provides an expressive programming paradigm. In a functional language such as **miranda** [21], the user can introduce powerful operators, together with variables of sophisticated higher-order types that can be defined by very general formulae. A **miranda** script is a definitive system that determines the transition from one **miranda** environment to another associated with changing the values of explicitly defined variables in the script. Referential transparency is a central principle of **miranda**, and editing the script is outside the scope of the functional programming paradigm. Within the **miranda** system, the effect of redefining explicitly defined variables in the definitive system can only be imperfectly realised in other ways. One approach is to omit such variables from the script, and to obtain the values of other dependent variables through function evaluations with different parameters. Another approach that potentially represents state more faithfully involves introducing variables whose values are histories.

A definitive system is a significant generalisation of a functional programming script. Editing the script falls within the definitive programming paradigm, and the redefinition of a variable is interpreted as a transition from one computational state to another. Redefinition permits dynamic modification of the environment for evaluation. In the simplest cases, this entails redefining an explicitly defined variable, but it can also serve to set up a definitive system to represent new transitions, or to add new definitions. This effectively means that the state transition model above is enriched by introducing other transitions corresponding to the introduction of new variables, and to the redefinition of implicitly defined variables. Note also that variable redefinitions can refer to the values of variables in the current state. This generalisation may appear to be a matter of convenience where user-computer interaction via a definitive system is concerned, but has wider implications.

From the functional programming perspective, perhaps the most unusual aspect of developing definitive systems in the manner described above is that the conventional distinction between constructing and executing a program is blurred. In introducing new variables and definitions, the user is incrementally specifying an environment that can be explored through variable evaluation after the manner of functional programming, but may exercise an additional freedom to interpret a redefinition as a state change within a single environment, rather than as a new choice of environment. From the definitive programming perspective, as will be illustrated in the examples that follow, the distinction between transitions from one computational state to another has to be understood with reference to privileges of agents. Pure functional programming is a restricted form of definitive programming in which the user establishes a particular computational state by editing a script, and authorises no agent to change this state subsequently.

Definitive programming on the above pattern is limited to identifying sequences of state transitions that have a useful interpretation e.g. via the observed behaviour of an object. General-purpose programming demands much more than user-driven sequences of state transitions can provide. In modelling the behaviour of a complex object, the state transitions that are permissible will depend upon context. In a complex application, several different kinds of state changes occur concurrently. The computational model underlying general-purpose definitive programming is supplied by the "abstract definitive machine (ADM)". In this model, the user is generally one amongst many agents that can perform state transitions by redefining parameters within a definitive system. Reference to values of variables in the current state is essential when specifying agents in the ADM.

The ADM has been described in detail elsewhere [8], and only its most relevant features are considered here. During the execution of the machine, the computational state is represented by sets of definitions and actions D and A respectively. The sets D and A change dynamically as the program executes, through redefinition in D as prescribed by actions in A, and through the introduction or deletion of definitions and actions. A program for the abstract definitive machine comprises a set of entities, each of which specifies a set of definitions and a set of actions that are

to be instantiated or deleted as a whole. To initiate execution, an appropriate set of entities is instantiated. Execution then proceeds in a sequence of machine cycles in which several actions may be performed concurrently.

Each action in A takes the form of a guarded sequence of redefinitions and/or instantiations or deletions of entities. The guards in actions are boolean expressions in the variables that appear in D. On each machine cycle, these guards are evaluated in the context supplied by D. Provided that there is no interference, the corresponding systems of redefinitions and entity reconfigurations are performed in parallel. The most significant forms of interference occur when a redefinition results in circularity, or leads to the evaluation of a variable that is currently directly or indirectly being redefined. The definitions in D play a similar role to the definitions underlying a multi-user spreadsheet - they record the current state of the system as established and modified through actions. The actions in A correspond to the user dialogue actions. The ADM supplies a framework within which many different agents can act concurrently by redefining explicitly defined variables in a definitive system. Though the fundamental concurrent computational model is synchronous, it is possible to simulate asynchronous activity within the ADM by introducing appropriate control entities (c.f. [7,8]).

Definitive programming has many potential merits for design support:

1) it provides a state-based programming paradigm;
2) it allows agent privileges and actions to be explicitly modelled;
3) it supports rich techniques for data representation and presentation.

The remaining sections of the paper examine the prospects for applying definitive programming principles to the motivating design support issues introduced at the outset. For this purpose, the computational tools within the design support system that are not directly accessible to the designer will determine the data types and operators in the underlying algebra. This means that, below a certain level of abstraction, computation in the design support system is conceived in conventional functional programming terms. How far the computation within the system is recast in definitive programming terms will reflect the extent to which the designer can penetrate the system usefully. For the present, even the degree of designer involvement conceived below may prove impractical. In principle, the intelligent intervention of the designer could be valuable even in some low-level computations, e.g. in resolving problems involving geometric singularities for purposes of graphical display.

§2: Definitive principles and the design process

How are definitive programming principles intended to be used for design support? This issue will be discussed abstractly in this section, with cryptic reference to the problems of designing simple mechanical objects such as doors that will be considered in greater detail in §3.

Three aspects of the design process in which interaction with the designer plays an essential role can be identified:

1) representation of the design object itself, and validation of the design

The designer typically has to construct a model of an object to be manufactured, to simulate its behaviour in the intended application under reasonable assumptions, and to confirm that this behaviour is consistent with that required. This part of the design process involves a cognitive component that only the designer can supply.

2) representation of the designer's concepts of the design object, and their development

To aid the designer, it will be necessary to impose some conceptual structure upon the design object, for instance, in the form of parametrisations that allow features to be conveniently modified. The appropriate form for these parametrisations cannot be preconceived, since they reflect the peculiar characteristics of the object, and the designer.

3) representation of the design object by and to the designer through the computer interface

The designer will need to be able to adapt the protocol by which data about the design

object is presented dynamically, and to determine the manner in which feedback about the current form and status of the design object is given. These will depend upon the specific design problem, and must be guided by the imagination of the designer. A standard repertoire of input and feedback techniques can support only very limited applications.

Amongst the three, 1) is of course the primary concern, from which 2) and 3) are derived. A good designer will choose parametrisations of design objects to suit the intended function of a design object, and modify these to reflect the results of simulation. To work effectively, a designer will need to construct environments within which different parametrisations of a design object can be conveniently formulated and flexibly invoked, and within which the necessary processes of validation can be reasonably faithfully and efficiently performed.

At any stage in the design process, the current status of the system with respect to 1), 2) and 3) will have to be represented. The principal end-product of the design process will be the representation 1), but there may be a significant role for variant designs associated with 2), and for an interface 3) that provides a realisation for the abstract model described in 1).

Following [5], all three aspects of the design process considered above will be addressed using definitive principles. ADM models will be used to express the current status of the system with respect to 1), 2) and 3). The same paradigm for representing states and transitions applies in all three contexts, viz the configuration of a definitive system to suit a proposed transition followed by appropriate changes of explicitly defined variables. As has been explained above, the designer will be privileged to set up these models to suit the purpose of the design problem, and in this sense will in principle be free to intervene in all three. The agents that typically initiate changes in state within these models have a different significance however, as will now be explained.

Definitions and actions that represent the design object

The model of the design object, as produced by the designer, will be an ADM program that describes characteristic relationships between parts of the object, and expresses the ways in which these can change as the state of the object changes. For a very simple object, the ADM model might consist of a single definitive system, together with actions that express the conditions under which explicitly defined variables can be changed (e.g. the conventional door). For more complex objects, a program that makes fuller use of the capabilities of the ADM is required (e.g. the door that closes automatically under the action of a spring). Transition within the design object model is to be understood with reference to the intended use of the object, so that the ADM actions will correspond to changes in state that might be initiated by design object daemons representing agents acting in the application (e.g. a horse closing the upper half of a stable door).

Constructing a design object model involves a cognitive process, whereby the designer formulates definitions and actions to represent the observed or intended relationships between components of a physical object (e.g. describing the locus of the lock of the door as its aperture varies). It cannot realistically be a comprehensive model of the behaviour of the design object (e.g. several people using a door). The validity of a design object model can only be assessed, to some extent, by analysis and simulation. The ADM model of a design object provides the basis on which appropriate simulations can be built. It is intended to provide a computational model within which the designer can intervene to resolve conflict if necessary (e.g. to specify a priority for door users).

Definitions and actions that assist the designer in developing the design object model

Determining the form of the design object model is choosing a representation for the design, and impinges upon design support only in so far as some representations are better than others for purposes of analysis and simulation. Central to the designer's view are the methods within the design support system that make it possible to redesign an object i.e. to change the object in ways that are outside the scope of normal use. These will be represented by ADM models that most closely represent the designer's perspective on the design object, in which the definitions are expressly under the control of the designer. The simplest design object manipulation models are parametrised objects: definitive systems that describe several different designs depending upon the choice of certain explicit parameters. Other definitive systems will establish relationships between specified objects at different levels of abstraction (e.g. ensuring that all the doors to a building

follow the same pattern), and can be used to maintain appropriate constraints (e.g. to relocate the hinge of the door whilst maintaining the correct geometry for the door cavity in the wall). Actions in this context might serve to transform the designer automatically from one framework for manipulation to another (e.g. switching from a context that relocates a hinge without changing the dimensions of the door, to one in which the width of the door is changed), or to impose constraints upon the designer (e.g. automatically revoking redefinitions that violate a specified constraint).

Definitions and actions that support the realisation of design object

Definitive principles are particularly well-suited for establishing direct connections between one data representation and another. The need to represent the design process to the designer effectively will be met by regarding the interface to the designer as itself a definitive system whose value is specified in terms of the internal ADM models described above. The definitions and actions that comprise this design object interface model form an essential part of the user-interface to the design support system. The designer is intended to adapt this model to suit the specific design task, but it will be preprogrammed to a much greater extent than the internal ADM models, and will generally perform most transitions through autonomous redefinition. The role of the design object interface model is to ensure that the definitions that describe the current screen layout properly reflect the nature and status of the current interaction. For instance, if the behaviour of the design object is being simulated, a set of definitions defining the external representation of the design object in terms of the internal representation will be established. If the designer is currently entering new information about a design object, a set of definitions to describing the appropriate screen layout, to include for instance menu format and status, will be set up.

In such a view, the designer's role is the development of an ADM program that combines several computational strands, resembling the musical counterpoint between independent voices. Closest to the machine level are the activities that describe the screen manipulation and feedback to the designer. Within the design application, there are definitions to represent the design object, and actions to be performed by the design object daemons in simulation. There is the program that the designer develops in order to manipulate and transform the design object in the process of design itself. Whether the approach to design represented by such an ADM program effectively meets the requirements of IIICAD depends upon the scope for dynamic intervention and modification given to the designer. It may not be possible for the designer to suspend and redirect a simulation, or to modify the way in which a design object is presented, for instance. The goal of present research on definitive principles for design is to investigate how concurrent execution of these computations can be supported in such a way that the designer can effectively monitor them and intervene as appropriate.

§3: Illustrating the use of definitive principles

Suppose that an architect is designing a door to suit a particular building. The context into which such a door might be placed can be described in an existing prototype system thus:

```
line      n1 = [NW, Lframe]
real      tablelength
line      n2 = [Rframe, NE]
point     NW
point     NE = NW + 6*doorwidth
int       k = 1 + tablelength/doorwidth div 1

real      doorwidth
point     Lframe = NW+k*doorwidth
point     Rframe = NW+(k+1)*doorwidth

monitor   k>5
```

The above definitions are intended to assist the architect to relocate the door so that there is sufficient room for a table to the left of the door frame, and to keep the width of the door in constant proportion to the length of the wall to which it belongs. The definitions enable the architect to express the essential relationships even though the position of the wall is as yet

unspecified. Note that these definitions are for the benefit of the designer: they will determine values that are fixed when the door is in use. Monitoring the condition k>5 means ensuring that a message is displayed as and when the value of k exceeds 5. A monitored condition might be implemented by establishing a variable whose value is a string defined by the formula:

if k>5 **then** "Warning: k>5\n" **else** ""

and setting up a variable whose value was a window in which the values of all such monitoring variables was concatenated. This illustrates the nature of the definitions used to support the implementation.

Certain characteristics of the door, such as its dimensions and type, are to be determined by the architect, and thereafter fixed. Choosing these characteristics constitutes door design. To determine whether a particular design is appropriate, the architect will need to simulate the door in use. For this purpose, the architect requires a computational model of how the door will behave. A simple definitive specification of a conventional door, as it might appear on an architectural plan, is:

real width = doorwidth
bool open
line door = [hinge, lock]
point hinge = Lframe
point lock = hinge + **if** open **then** {0, -width} **else** {width,0}

In interpreting such a definitive system as a design, it is not enough just to consider the current values of the variables and the relationships between them. It is essential to appreciate the status of each definition, and take account of the ways in which particular agents can change the system state. In general, some definitions will reflect constraints imposed by physical laws. The definitions relating door, lock, and hinge reflect the physical characteristics of the particular type of door chosen by the designer. Other definitive systems would be characteristic of a revolving or a sliding door. Definitions may also reflect rules otherwise imposed upon the design, as when the width of the door is pre-specified. Within the framework of such constraints, definitions can be freely chosen by the designer to ensure that the the system exhibits an appropriate behaviour. The position of the door is determined by where the hinge is located; this cannot be changed in normal use. In simulating typical use of the door, only the value of the variable 'open' can be changed. Declaring the circumstances under which particular variables (such as the variable 'open') can be redefined by the users of the system is a part of the design.

Agent protocols are generally implicit in design. It is obvious to the architect that the orthodox room user cannot change the width of the door. Because definitive methods allow state changes associated with design and simulation to be represented in the same manner, they highlight the need for protocols. In the above example, "opening the door" and "moving the hinge" cannot otherwise be distinguished. (Compare a conventional simulation program in which "moving the hinge" would involve editing the program, and "opening the door" supplying further input.) A user protocol may also be needed to complement a complex design, as when a designer formulates rules for the use of the design object.

In order to express the design in terms of what agents can do, and when, a designer will wish to organise actions by agent, specifying the preconditions enabling each potential action. (This is the approach that is being developed in the notation LSD [7].) A simple agent action (such as opening the door) will correspond to changing the value of an explicitly defined variable in the context of a definitive system. A possible protocol for the door user might be:

not open -> open = True
open -> open = False.

A more subtle protocol permits the door to be locked by another agent:

not locked and **not** open -> open = True
open -> open = False.

There are many variants of these protocols, as when a door automatically locks on closing for instance.

A protocol does not comprise actions directly suitable for interpretation in the ADM. An action in the ADM is to be executed when its guard is true: during a simulation an agent performing actions permitted by its protocol may choose between several possible options, and perhaps perform no

action at all. The door protocol above gives no information about when a user is likely to open the door, nor what should happen if one users tries to open the door at the same time as another tries to lock it. The total concept of a design object is open-ended, and cannot be divorced from its intended use. The application of definitive principles permits the representation of a design object by an ADM program that encapsulates the characteristic behaviour of the object (e.g. how the door responds to being opened and closed). Such a program can then be used as a component in simulations of the object in typical use (e.g. the door being opened and closed by several users in a concurrent system). These principles are more fully illustrated by the next example.

Suppose that an architect wishes to specify a stable door, in which the upper part of the door can be opened independently of the lower. Such a door could be described by the definitive system:

bool uopen
bool lopen
line udoor = [hinge, ulock]
line ldoor = [hinge, llock]
point hinge
point llock = hinge + if lopen then {0,-width} else {width,0}
point ulock = hinge + if uopen then {0,-width} else {width,0}

In this model, a typical user can independently change the values of the boolean variables 'lopen' and 'uopen'. In practice, the stable door might be designed so that the upper door overlapped the lower. The variables 'uopen' and 'lopen' would then be constrained to satisfy

$$\text{lopen} \Rightarrow \text{uopen}.$$

Note that this constraint itself does not fully specify how the stable door should respond to movement of the upper and lower doors. It is consistent with the supposition that the stable door behaves like a conventional door, for instance.

The protocol for using the overlapping stable door is more complex. It must express the fact that if both upper and lower doors are closed, and the lower door is opened, then the upper door also opens. At this stage, it is then possible to close the lower door whilst leaving the upper open. This can be expressed by a protocol of the form:

not uopen -> uopen = lopen; lopen = true
uopen -> lopen = false

This protocol is to be interpreted as asserting that the definition

$$\text{uopen} = \text{lopen}$$

pertains whilst the lower door is being opened from the context in which both upper and lower doors are presently closed. Such a protocol provides the basis of an ADM program simulating the operation of the door.

As a more complex illustration of how a design object is described by an ADM program, consider the implications of attaching a locking device to the stable door so that the upper and lower doors become "ganged" (i.e. locked together). In this case, the user protocol includes the action:

uopen == lopen and **not** ganged -> ganged = true

that has more significant implications than previous redefinitions. Whilst the doors are unganged, 'lopen' and 'uopen' can be independently controlled as described above. Once the doors are ganged, the definitive context within which actions are performed is changed radically. Where before there might have been a variable 'open' with a value partially specified by the definition

open = **if** lopen==uopen **then** uopen **else** @,

the variables 'lopen' and 'uopen' are now more appropriately defined in terms of 'open', via:

lopen = open
uopen = open,

and the variable 'open' is subject to user control. In yet another model, the doors might become ganged after an appropriate locking device was set, on the first occasion on which the condition 'uopen == lopen' was satisfied.

The development of ADM models of design objects for purposes of simulation is as yet at an early stage. Constructing ADM models to simulate objects such as the stable door in use is a subject of current research beyond the scope of this paper. For a fuller discussion of some of the issues, the interested reader can consult [8,9], in which an ADM model of a small system comprising two blocks connected by a string moving in discrete steps under the control of independent agents is described. It remains to be seen to what extent the construction of such models can be simplified through the application of definitive principles, but there are some indications that the knowledge of data dependency implicit in the ADM model for action can assist in detecting interference between concurrent actions, and allow the designer to intervene to resolve them. For instance, in the prototype blocks simulation referred to above, the fact that moving the right block to the right when the string is taut invokes a definitive system in which the position of the left block is defined in terms of that of the right, and vice versa, enables the ADM to recognise interference between the actions. In the context of this paper, it is particularly significant that the simulation is gracefully suspended when such conflict arises, so that the user can be consulted to decide upon the appropriate action, and the simulation resumed.

§4: Prospects for applying definitive principles for design support

The aim of this paper is to put into perspective several aspects of the research into definitive principles to CAD systems represented in previous papers [4,5,6,7]. This research is focussed upon supporting interaction between the designer and the system in crucial aspects of the design process:

1) representing intrinsic relationships associated with design objects, and (potentially) simulating their behaviour. Relationships referred to here are characteristic of the function of the object;

2) assisting the designer in the manipulation of representations of design objects, support context switching between representations suited for different purposes. Here the relationships being modelled are those that are implicit in the designer's view - how the designer has conceived the structure of the design object;

3) allowing the designer to adapt the feedback from the system to suit the particular characteristics of the design problem. Relationships being modelled here relate the different types of data within the system, perhaps linking data values in entirely different semantic categories.

In each case, a role for definitive systems has been identified, together with a need for a framework such as the ADM supports to allow these definitive systems to be dynamically reconfigured, whether according to context, or under the control of the designer. Even autonomous reconfiguration may be programmable within the system, but should ideally be intelligible to the designer, so that it can be suspended and resumed after intervention by the designer. More research is needed into how the ADM model of computation must be presented for this purpose.

Logic programming methods figure prominently in alternative approaches to 1). Comparison between simulation in the ADM model (as represented in [8]), and the commonsense reasoning approach proposed e.g. in [13,14], raises many interesting issues. Some recent research has questioned the role of inference in cognition [15,16], and it will be of interest to consider whether definitive principles can offer a better cognitive model.The three classical problems for reasoning about action: the frame, ramification and qualification problems also deserve serious examination in this light. There is superficially a strong resemblance between the use of definitions and constraint-based methods [10], but there are significant differences. Whereas constraints are best viewed as assertions about the current state, definitions should be seen as giving information about latent change, and address the concept of relationship in conjunction with potential change.

There are interesting connections between definitive programming and both object-oriented and functional programming to be further explored. The present approach to supplying user-defined functions in the definitive notation DoNaLD promises to import the significant advantages of recursively defined functions, and of object-like abstractions [9]. The absence of data hiding principles in definitive programming distinguishes it from the object-oriented paradigm, and proves to be an advantage where the modelling problems discussed in [20] are concerned. Recent research

has involved the implementation of a definitive interface to the **miranda** system that makes it possible to redefine variables without invoking the editor. In such a framework, it is easy to construct a transparent state-based model of a conventional door, but apparently difficult to cope with a stable door, notwithstanding the potential for the use of higher order functions. These experiments strongly indicate the importance of some form of history sensitivity to complement pure functional programming methods [1,2].

The merits of definitive principles in respect of 2) have been well-established in prototypes. It is important to remember that the perspective of the designer changes in the process of design, and that the virtues of referential transparency can only be realised relative to the designer. It is in the process of helping the designer to maintain a computer representation of an object that is consistent with the designer's current view that definitive methods prove most obviously effective. As a particularly simple illustration of this concept, it is trivial to construct a mechanism for consistently pointing at an abstract point within the DoNaLD system. The importance of such reference capabilities has been discussed in detail elsewhere [9]. Definitive principles also promise to support the representation of objects at different levels of abstraction, and where information is partial. The hierarchical nature of agent privileges, whereby the designer works within a framework of definitions and actions reflecting physical constraints, and the designer frames the context within which the behaviour of the design object is simulated, implicitly gives support for views.

The ADM computational model has some features in common with a rule-based approach (c.f. [3,19,17]). The ADM model of computation is intended to help the designer to interpret the autonomous computation performed by entities, and enhance the prospects for intervention and cooperation. In this way, the functions of the intelligent supervisor in the IIICAD system [19] can perhaps be served by a program that relies upon the designer rather than a set of preprogrammed rules for its intelligence.

Communication within a CAD system makes essential use of symbols that reflect the mental structures in the designer's imagination. Definitive principles are very well-suited to establishing the direct links between one form of data representation and another. It would be easy to use the underlying **openshape** data structures in DoNaLD to create visual representations, for instance. Object-oriented methods have already proved quite effective in establishing the relationships between internal and external representations required for good interfaces [11], but there are problems in making these relationships appropriately sensitive to data values [12]. A major problem for a preprogrammed interface is that the most appropriate representations are hard to anticipate without problem specific knowledge. Similar difficulties are encountered in making user-adaptive interfaces. It is in these respects that the blurring of the distinction between program development and execution in the definitive programming paradigm referred to in §1 may prove most advantageous.

Concluding remarks

Current work on applying definitive principles to design support is focussed on two main issues: developing a geometric modelling package (CADNO) following the preliminary design proposed in [5], and developing ADM models of design objects for simulation purposes [8].

The informal description of definitive programming in terms of a state-transition model in §1 invites direct reinterpretation in design terms. States in which the set of values includes undefined values are of limited interest in programming, but can have valid meanings in design. The confusion between the concepts of editing and executing a program is entirely appropriate in the context of design, where the distinction between state changes of the design object associated with design and simulation is essentially a matter of perspective (contrast 'relocating a shelf' with 'opening a window'). The flexibility afforded by the use of definitive principles is directly related to the way that design and simulation are integrated in this fashion. In principle, it allows the designer to assume - or to delegate to appropriate agents - arbitrary privileges to reconfigure a design or the design environment without moving outside the system. Such an approach allows the designer to adopt a "Wrong-Every-Time" rather than a "Right-First-Time" methodology, in the spirit of the retrospective planning of [18]. Experience with prototypes has so far confirmed the importance of these virtues.

Acknowledgements

I am greatly indebted to Steve Russ for helpful criticisms of earlier drafts of this paper.

References

1. Abelson H, Sussman G J *Structure and Interpretation of Computer Programs* MIT Press 1985
2. J Backus *Can programming be liberated from the von Neumann style?* Turing Award Lecture 1977, CACM **21**, 8 (August) 1978, 613-641
3. P Bernus, P J W ten Hagen, P J Veerkamp, V Akman, *IDDL: The Language of a Family of IIICAD systems*, in Intelligent CAD Systems 2: Implementation Issues, Springer Verlag *to appear*
4. W M Beynon, *Definitive Principles for Interactive Graphics*, NATO ASI Series F:40, 1987, 1083-1097
5. W M Beynon, A J Cartwright, *A Definitive Programming Approach to the Implementation of Intelligent CAD systems*, in Intelligent CAD Systems 2: Implementation Issues, Springer Verlag *to appear*
6. W M Beynon, A G Cohn, *Representing design knowledge in a definitive programming framework*, (paper prepared for IFIP WG 5.2 Workshop Cambridge, UK September 1988)
7. W M Beynon, M T Norris, M D Slade, *Definitions for modelling and simulating concurrent systems*, Proc IASTED Conf ASM'88, Acta Press 1988, 94-98
8. W M Beynon, *Definitive programming for parallelism*, Univ of Warwick RR#132, 1988
9. W M Beynon, *Evaluating definitive principles for interactive graphics*, Proc CGI'89 *to appear*
10. A Borning, *The programming language aspects of ThingLab, a constraint-oriented simulation laboratory*, ACM Transactions on Programming Languages **3**(4), 1981, 353-387
11. A Borning, R Duisberg, *Constraint-Based Tools for Building User Interfaces*, ACM Transactions on Graphics, Vol 5 No 4 October 1986, 345-374
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. M L Ginsberg, D E Smith, *Reasoning about Action I: A Possible Worlds Approach*, Artificial Intelligence 35 (1988) 165-195
14. M L Ginsberg, D E Smith, *Reasoning about Action II: The Qualification Problem*, Artificial Intelligence 35 (1988) 311-342
15. P N Johnson-Laird, *Mental Models* CUP 1983
16. D McDermott *A critique of pure reason* Comput Intell **3**, 151-160, 1987
17. R Popplestone, T Smithers et al, *Engineering Design Support Systems*, IKBS/MS 7, 1986
18. T Takala, *Design Transactions and Retrospective Planning Tools for Conceptual Design*, in Intelligent CAD Systems 2: Implementation Issues, Springer Verlag (to appear 1988)
19. T Tomiyama, P J W ten Hagen, *Organization of design knowledge in an intelligent CAD environment*, CWI Report CS-R8720, 1987
20. T Tomiyama, *Object-oriented programming for intelligent CAD systems*, in Intelligent CAD Systems 2: Implementation Issues, Springer Verlag (to appear 1988)
21. The **miranda** manual, *Research Software Ltd*, 1987