Definitive principles and the specification of software

W M Beynon, M D Slade, S B Russ, Y P Yung, Y W Yung
Dept of Computer Science, University of Warwick, Coventry CV4 7AL

Introduction

This paper is the first of two concerned with the prospects for developing a new programming paradigm ("definitive programming") that is essentially based upon the principle of variable definition illustrated in its simplest form in the spreadsheet. The paper is in part an interim report on the broad programme of research that has been carried out so far on the definitive programming project. In the interests of brevity, some familiarity with the basic ideas, as set out in references such as [6,7,8,9,10,11,12,13,14], is assumed. The companion paper [10] is the most appropriate single source, since it illustrates the method of software specification considered in this paper with reference to the techniques and prototypes described elsewhere.

When proposing and justifying a new programming paradigm, there are several important issues that should ideally be addressed:

1) clear and formal description of the principles that define the paradigm;
2) practical demonstration of how the principles can be applied in general applications;
3) characterisation of the programming paradigm in relation to other approaches.

Until a programming paradigm is well-established, issues 1), 2) and 3) are best explored in parallel. Even for many established paradigms, these issues remain a focus for ongoing research. The generality of functional programming methods in practice is controversial, for instance, as is the formal characterisation of object-oriented programming.

Research on definitive programming has so far made considerable progress in respect of each of the three issues above. The representation of state-transition systems using families of definitions (cf ¤3 below) is - informally - a characteristic principle that demonstrably has special qualities in connection with interactive systems and incremental design [6,7,8,12,13]. The extension of such models for user-computer interaction to encompass several agents interacting in a computation has potential as a means of modelling and simulating concurrent systems [9]. As formalised in the computational framework of the Abstract Definitive Machine [14,11], definitive programming supplies an unusual parallel programming paradigm that is sufficiently versatile for the implementation of CAD software [7, 8,13].

The present status of our research is not such that a categorical answer to the question "what is definitive programming?" can be given. In our view, there are strong indications that there is a distinctive style of programming - probably richer than that encapsulated in the ADM - for which issues 1), 2) and 3) can be satisfactorily addressed. This paper and its companion paper [10] are both concerned with the nature of general-purpose definitive programming in so far as it is at present understood. In this paper, we argue that the fundamental concerns of software specification point to the need for programming methods (such as we have developed during the definitive programming project) with good characteristics for interactive and incremental use. As a complementary exercise [10], we show that definitive principles can be effectively applied to software specification and prototyping.

¤*1 The nature of specification*

What is a specification? Following Turski and Maibaum [21], we may adopt the view that a specification is a formal expression of the intended meaning of a program that admits a dual interpretation. On the one hand, it is an accepted description of the problem to be solved; on the other, it is a prescription for the program that solves the problem. A specification describes the intended meaning of a program in terms that are oriented towards the application; a program in terms that are oriented towards a calculating machine.

The arguments concerning the relationship between programs and specifications advanced in [21] support the view that "a specification" and "a program" cannot be clearly distinguished. The specification differs from the program through being "at a different linguistic level": indeed, "the only reason to construct a program is to express the intended meaning in a different way" ([21] p8). Program development is seen as the process of translating a formal specification into a program:

"The availability of compilers, and their, by and large, acceptable quality determines a demarcation line between what is and what is not considered development activity ... this line is artificial, and merely represents the commonly accepted limit of strict calculability of linguistic levels." ([21] p123)

There are clear dangers in too simplistic an interpretation of the equivalence between a "specification" and a "program". If indeed a specification is no more than a very high-level program, in what sense can the problems of creating an appropriate program be alleviated by first developing a specification? This is of course an old and fundamental problem, sometimes known as the "second box problem". In approaching this problem, it is important to appreciate that a specification cannot be "correct" in the same way that a program is "correct relative to a specification", but can only supply a description of the required behaviour of the system to be developed that can be validated by observation in essentially the same limited way that a scientific theory can [21]. In so far as a specification is a formulation of the required behaviour in application-oriented terms, this validation process will be simpler than the direct validation of a machine program, but this is no reason for supposing that the principles for constructing a valid specification are clarified as a result.

Our conception of specification assumes some basic premises. Whilst endorsing the view expressed by Turski and Maibaum concerning "the specification level as the point of unification of formality of expression and parting with the implicit semantics" [21], we put our emphasis upon the derivation of a specification as an incremental process of formalisation and validation involving complex interactive and iterative design. Such a view of the specification process is particularly well-represented in the work of Balzer, Goldman and Sartout [4,20], which emphasises the essentially dynamic nature of the correspondence between "program" and "application", and the need for a style of programming that supports both the clear articulation of the program, and of the salient aspects of its derivation. In this context, we have no distinction between specification and program in mind; for us, the specification is to be a very high-level program, and our central concern is with how it is to be constructed.

The main objective of this paper is to advocate novel programming principles that may be particularly well-suited as a basis for specification languages. Clearly - in the spirit of Glaser, Hankin and Till, who contend that functional methods support an application-oriented view of programming [15] - we shall need to demonstrate that our principles have the generality and expressive power to support the representation of system behaviour at a very high level of abstraction. At the same time - in view of the issues raised above - we must be primarily concerned with formulating a programming style that can directly assist the process of constructing and validating programs. Notice that this presumes a concept of programming that is in one respect much more general than is customary. Our presumption is that our programming notation can support what from a behavioural perspective would normally be regarded as "incomplete" programs, viz the fragments that are introduced in the process of arriving at an initial specification. It may even be fair to say that - at least for sequential programs - our interest in the issues of representation end at the point (should there be such a point!) at which "the specification is a complete description of the intended behaviour", when our program can be readily re-interpreted in whatever is the most favoured paradigm, be it functional or object-oriented programming. Note here that - because of the application-independent nature of the specification process - we are only concerned with programming paradigms that are potentially well-suited for specification.

Our analysis points to the conclusion that a good paradigm for specification will support incremental programming in which interaction and iterative design play a significant role. For the validation of such a specification to be conveniently performed, it will also be desirable for the specification to provide a cognitive model (c.f. [4] Principle #5). A concern with "how easily a program can be developed" is clearly very different in nature from a concern for "how comprehensibly can the intended behaviour of the application be represented in cognitively appropriate terms". It is nonetheless plausible that any computational paradigm purporting to cognitive validity should be one that readily supports revision and extension of a program; after all, in our mental processes we are presumably constantly involved in adaptive specification and validation in a very volatile environment.

This paper is a development of the above ideas. ¤'s 1, 2 and 3 examine particular aspects of existing approaches to high-level programming as they relate to our view of specification. ¤4 considers the resemblance between state changes in design and simulation that motivates the novel approach represented by definitive ("definition-based") programming outlined in ¤4.

¤2 Appropriate programming paradigms for specification

Our primary objective is to devise a programming paradigm that supports incremental program development, representing the design process to the user in such a way that the current status of the design, and the possible options for enhancement and modification are as transparent as possible. In this context, "the design process" may refer either to 'constructing a specification for a system whose actual behaviour is already established and observable' (c.f. [16]), or - as is of greater practical interest - 'developing a specification that describes an intended behaviour that is not entirely preconceived'. In the terminology of Turski and Maibaum [21], we seek a permissive style of specification that deals with incomplete information in a non-singular fashion. Our concern is with executable specifications, so that a program should serve both as a abstract mathematical object intelligible to the user, and also as a prescription for a computer, interpretable in an appropriate machine model.

We shall first clarify what we mean by "a programming paradigm that supports incremental program development". It may be argued that there is a proper separation of concerns between 'the programming medium in which we choose to describe the behaviour of a system' and 'the design methodology we should apply to construct a specification'. The crux of our case is that the programming media that have so far been adopted for describing system behaviour (be they conventional procedural, object-oriented, declarative or specification languages) are in some respects not well-suited for incremental program development. In other words, however we choose to develop large programs using these paradigms, we shall need very sophisticated auxiliary design tools. On what grounds can we make such a strong claim? Because - on the one hand - it seems that the primitive methods available to a designer wishing to modify a program within one of these paradigms (e.g. such as defining a new constraint in a constraint-based system, or defining a new function in a functional programming system) are often associated with very obscure changes in the status of the design, whilst - on the other hand - relatively commonplace steps in the design process (e.g. such as establishing or modifying relationships between objects in the application, taking account of more characteristic physical properties of the environment) typically necessitate a sophisticated process of editing the program text (such as - to take a trivial example - is involved in adding a parameter to a procedure in a Pascal program).

Note that in viewing a programming notation from the perspective of "appropriateness for program design" we are looking beyond the interpretation of a static program (i.e. the program as a mathematical object, as opposed to a recipe for a calculating machine); we are concerned about how easily we can change the program as a 'dynamic object' in the sense of [4]. We shall judge the quality of the specification medium by how easily we are able to develop an incremental specification that is faithful to the designer's expressed intentions at each stage, and can be legitimately partially validated through simulation even though it is as yet incomplete. Of course, our concern is with what we can achieve in principle within a chosen programming paradigm - there can be no magic medium whose use guarantees good representation of the design. We must also distinguish between "making it easy to implement changes" (i.e. through developing better editing or compiling tools), and "making it easy for the user to apprehend what changes can / should be made".

An example may help to illustrate the issues involved in the development of a specification. Suppose that we wish to model the characteristics of a room to be used by several occupants for various functions. Our completed specification will take the form of a very high level program with a behaviour that corresponds precisely to the observed - or perhaps the intended - behaviour of the room in use. We need not distinguish between specifying the characteristics of a given room, and those of a room to be freely designed - in either case, our task involves similar processes of formulation (i.e. representation in our chosen specification notation) and validation (i.e. checking through simulation that our specification exhibits behaviour consistent at each stage with the observed / intended behaviour of objects in the application). For preference, we shall consider the more complex and typical use of specification associated with "room design".

We must of course take some primitives as given. For simplicity, let us assume basic geometric primitive functions, such as line length, point intersection of lines, lines incident with points etc., and objects comprising sets of points, lines and planes. Typical design issues are:

a) the preliminary specification of characteristics of the room, such as "the room is cubic";
b) the form of objects to be disposed about the room e.g. tables, desks, lamps, power points;
c) the form of special features of the room e.g. sliding / hinged doors;
d) the number and location of these features e.g. how many doors, and where they are;
e) constraints such as "at most two doors", "no overlapping items of furniture";
f) relationships such as "all tables are identical", or " the doors are symmetrically disposed".

During the process of designing the room, some simulation for the purposes of validation of the (partially completed) specification might be required. For instance:

A) relocate a table, change the aperture of a door, move the lamp, use another power point;

B) remove or introduce an item of furniture;
C) simulate the movement of a table, taking account of geometric constraints;
D) simulate the actions of two independent agents acting concurrently in the room;
E) explore the implications of changing the perceptions or privileges of agents.

We should not be in any doubt that supporting such a design process faithfully would be a difficult task within any programming paradigm. In B) for instance, we have major research problems concerned with "qualitative reasoning" to resolve if we are to devise a satisfactory environment for validation. After all, the possible ways in which a lamp placed upon a table might respond to the movement of the table are neither predictable nor easily enumerated (c.f. [12]). What is more, we should not think of the design process as proceeding in a carefully preconceived fashion from one consideration to the next: on the contrary, a designer will be free to reconsider a design decision at any stage, and - for instance - may decide to replace three identical tables that cannot overlap each other by a nest of tables that can all occupy the same floor space. The fact remains that the real world application we have chosen is by no means improbably complex, and developing a specification typically involves considerations of the kind described above.

The implications of viewing specification as encompassing a design problem are far-reaching. To discuss the issues fully is beyond the scope of this paper; for an excellent account of many of these see Balzer and Goldman [4]. To appreciate the difference in outlook that the specification of the program as a dynamic object entails, it is enough to contrast conventional algebraic and model-based approaches to specification with the principles of specification set out in [4] ¤3. The programming paradigm introduced below is intended to realise some of the objectives of [4].

A few general observations about specification are relevant to our present concern. Many existing approaches to specification exploit forms of implicit information whose interpretation requires inference. The potential virtues of implicit methods for representing incomplete information are clear. In the present context, there are dangers in expressing implicit information in the form of global assertions about actions and states, since this relies upon knowledge of the entire specification. The limitations of such methods in respect of simulation and cognition are well-represented in [18] and [19].

Constructive techniques for specification that favour the use of explicit representation, such as the use of an object-oriented programming paradigm, can avoid these problems, but only in general at the expense of disguising global unifying properties that simplify the articulation of the design. The need for both views is reflected in state-of-the-art design support systems that combine powerful inference techniques and process-based models in an object-oriented idiom [5]. Our integration of implicit and explicit methods of data representation is at a lower level of abstraction, and is centred around the concept of data dependency.

¤*3 Representing state and state transitions*

The need to support the program design process self-evidently presumes some method of representing state, at least in so far as the designer must be able to ascertain the current status of the design. The subtle interrelationship between the status of a partial design, and the state of the associated partially designed system in simulation, will be explored more fully below. It will be helpful to first examine the formidable semantic difficulties that the concept of computational state invokes, and explain how we propose to address them.

As the semantics of procedural programming languages illustrates, state representation typically proves problematic. The idea that the state of a system should be represented through the current values of a family of procedural variables is very plausible, but conventional ways to handle procedural variables lead to conceptual complexity. Suppose, for instance, that we represent the positions of objects within a room by recording the coordinates of the hinge of the door, the angle of aperture of opening, the position of the centre and NE corner of the table etc. In a naive procedural program that manipulates these values there is no convenient way to eliminate assignments to procedural variables that are inconsistent with the intended semantics of an object, so that e.g. we may move the centre of the table SW without relocating the NE corner.

Of course even such a naive program conceptually comprises composite sets of several reassignments (such as might relocate NE corner and centre of table, and the other corners and edges of the table in a physically consistent manner). Thus "moving the centre of the table alone" may resemble an atomic action as it appears in the implementation, but is not part of the user's conceptual view. This motivates the two traditional approaches to solving this problem: object-oriented programming, that makes it is possible to associate families of reassignments so that consistent updating is readily guaranteed, and the use of assertions, that makes it possible to annotate the consistent states ie to explicitly distinguish the logically interesting states, such as correspond to consistent locations of the elements of the table.

Many complex issues are involved in the apparently simply idea of "identifying consistent systems of updates", especially where concurrency is involved. Sometimes there is a direct relationship between an object and its parts (as in the opening of desk drawer), at other times, a loose relationship (as in a game of musical chairs). In general the effect of an action is context dependent (as when two tables touch), and one object may operate in different modes (e.g. a stable door that can be configured to open at the top or the bottom independently). It may also be necessary to describe synchronised action at a distance (as when the lights fuse).

Our approach to procedural programming is based upon one primitive concept. Characteristic of models based upon families of procedural variables are distinctive "systems of functional relationships between these variables" that are associated with particular changes of state. For instance, when we move the lower limb of an anglepoise lamp, the rest of the lamp moves according to predictable functional relationships. It is important to note how subtle these relationships can be: compare the functional relationships associated with moving a lamp when it is fixed on a table rather than placed on a table, the different consequences of moving touching tables in various directions, and the various ways in which the stable door can respond to movement of its upper and lower components depending upon its state. We choose to represent these functional relationships using definitive systems, that is, families of variables of appropriate types whose values are either explicitly defined, or implicitly defined by algebraic expressions in terms of the values of other variables and constants. The central abstraction of definitive programming is then "the functional relationship between variables appropriate to a particular action within a state". Note that a definitive system may describe a transition faithfully over a minute range e.g. provided tables aren't touching, though they be arbitrarily close.

It is of interest to consider definitive systems with reference to other programming paradigms. In object-oriented programming (OOP), mechanisms exist to implement such systems of functional relationships between objects readily, but they are very easily abused. Even such mechanisms -primarily designed with simulation and modelling in mind - have their limitations in respect of clear and semantically unambiguous representation of indivisible procedural actions. Whilst it is convenient to model many instances of tables in OOP, it is not so clear how to model two touching tables. We can introduce a message passing protocol to represent the effect of a single atomic action, viz the simultaneous movement of two touching tables, but once it becomes necessary to invoke message passing between objects to establish data dependencies conceptual control is lost. If this seems too pedantic a position to adopt in respect of sequential programming within the OOP, it is certainly a reasonable concern where concurrent actions of agents may be involved, as the semantic difficulties faced in parallel OOP models illustrate. How should we deal (for instance) with two independent table-movers manipulating touching tables?

There is superficially a close relationship between our approach and constraint-based programming, but our concept of 'a definition' should not be interpreted as 'a constraint'. This is not merely because a definition establishes an acyclic data dependency. There is a very important distinction between an equational view, and the use of definitions: an equation does not specify how a constraint is to be maintained. What is more, it is most appropriate to view a definitive system not as describing existing relationships between values within a state, but as expressing latent relationships invoked whenever an action is carried out. That is, definitive systems are primarily intended to represent actual and possible transitions, rather than states. Though it may be sometimes be more faithful to the application (or more efficient in an implementation) to presume that a functional relationship between variable values persists irrespective of what actions are to be performed, the state can in principle be represented simply by a system of values. In a more formal setting, it may be that whereas an equation prescribes that "a certain relationship holds in all interpretations", a definitive system expresses a specific way in which to "change the interpretation" whilst preserving a relationship.

Paradoxically, the dissociation of definitive principles from the representation of state establishes a link with the programming paradigm that has been most hostile to the concept of state. Pure functional programming may be seen as the simplest form of definitive programming, in which one agent (the user) acts through a single definitive system, viz that articulated by the system of functional relationships established through formulating a system of functions and re-evaluating these functions with different arguments. This is an adequate view in so far as the user can anticipate the effects of actions and articulate them unambiguously in functional terms without reference to a current state. Such a view does not in general reflect the need to admit other agents, as when one agent can perform several different actions according to a context beyond its control. This is not to ignore the possibility that we can sometimes simulate several agents and context-dependent actions by using higher-order functions (cf [17]). Perhaps - in the case of the touching tables for instance - we can discern higher order functions to define the functional relationships appropriate to particular actions in terms of the abstract location of the tables, but it will still be necessary to describe "the current location of the tables" - a state-based concept that requires reinterpretation in a referentially transparent framework. We can likewise describe the actions that an agent can perform in relation to a door that may or may not be locked in such terms when the agent holds the key. If on the other hand the status of the door is beyond the agent's control, it seems difficult to sustain a purely functional paradigm (c.f. Abelson and Sussman [1]). Similar concern about the limitations of a purely functional programming style is expressed by Backus [2]:

> "The primary limitation of FP systems is that they are not history-sensitive
> ... a system cannot be history-sensitive unless it has some kind of state".

¤4 Status of the Design vs State of the Simulation?

We have implicitly argued for a state-based paradigm for computation that can serve both to represent the status of the design, and the system state for purposes of simulation. Definitive principles have been shown to be relevant to the simulation process; it remains to show how they can be related to the design.

The key idea can be illustrated with reference to the design problem outlined above. For clarity, let us imagine that an architect is showing us around a room that is in the process of construction. If it should happen that there is a doorway but as yet no door, we do not suppose that this is part of the design. Nor would be make this assumption if the door were lying on its side against a wall, prior to installation. If the architect shows us that the window opens by sliding from left to right, we may appreciate that the window has been designed to open in this way, but do not interpret the action of opening the window as a change in the design. If, on the other hand, the architect indicates two alternative locations in which a built-in shelf can be mounted, we will interpret the two resulting states of the room as two alternative designs. In an idealised world, such as can be realised only through computer simulation, the architect might wish to exercise other options; to indicate the implications of relocating the window, or changing the dimensions of the room. As it is, the architect will sometimes appeal to such an idealised world of the imagination, as in indicating a possible site for a staircase when there is as yet but one storey.

Our illustration shows that distinguishing between changes in the state of the room that are associated with design and those that are associated with simulation is essentially a matter of interpretation. As room designers, we contemplate all kinds of state changes that do not correspond to physical transformations we could conveniently perform, but there appears to be no reason in principle why we should wish to represent these differently within an abstract specification of the design. The fact is that we can easily adjust the angle of aperture at which a physical door is set ajar, but could only modify the width of the door, or relocate the hinge, with considerable difficulty. Nonetheless there seems to be no good reason to choose to represent these possible ways of manipulating the door in different ways in a simulation program: after all, we might design a door for which the angle of aperture was fixed, and that could be opened by adjusting its width.

In conventional specifications, it is not untypical for changes of state associated with design to be represented in a totally different way from those associated with simulation. Changes in the design are associated with editing the program, whilst simulation is performed by executing the program. Such a convention for representation presumes knowledge of "the final specification"; the integration of design and simulation is hindered in consequence. For just such a reason no architect develops a preliminary design in bricks and mortar. The integration of design and simulation we are proposing effectively invokes the concept of a program that is able to edit its own text, and brings to mind Backus' observation regarding the limitations of a purely functional style [2]:

> "no program is able to alter the library of programs".

This is not to suggest that the distinction between design and simulation is insignificant - only that the distinction should not be reflected by how the associated state changes are represented. Practicalities of room reconstruction apart, the true distinction between "designing a new room" and "changing the state of a predesigned room" is bound up with a concept of how agents are privileged to act in relation to a room. As room designers, we are entitled to prescribe any changes to the state of the room we desire, subject only to the higher authority imposed through physical laws. As responsible occupants of a room, the repertoire of state changes we can perform is limited to those actions for which the room has been designed. The significance of this interpretation of design is nowhere more apparent than in surrealist art, where the conventions of functional room design are violated, and the concept of a room in which the door hangs from the ceiling is not incongruous.

For the purposes of simulation, at least in a sequential setting, it suffices to consider the actions that can be performed in a given state. To address the issues of design and simulation more fully, we shall also need to be concerned with agents. On the one hand, we shall need to simulate concurrent action, and so consider under what circumstances actions can be performed simultaneously. On the other, we shall need to interpret the status of the design in terms of the actions that particular agents can perform in particular states.

The above discussion motivates an agent-centred view of computation, in which we explicitly represent the state changes each agent is privileged to perform. Our programming paradigm is framed around this conception. By using definitive systems, we are able to describe such state changes with clarity and expressive power. By introducing an appropriate abstract machine model to formalise the possible interactions between the actions of different agents, we can represent the "status of the design" in terms of "what actions agents can perform in which contexts". Note that this directly associates the design with an "intended functionality" with respect to subscribing agents. For example, possible privileges for an agent might be:

I can open / close the unlocked door
I can open / close the door, whether it is locked or not
I can open the door if it is unlocked, I can close it if it is open.

Note that in the latter case, if I don't have the key, I cannot predict when I will be able to open it. As explained above, it is not essential - nor in general desirable - to suppose that the current state of the system is formulated using definitions (as opposed to families of explicitly defined variables), though this will sometimes be appropriate. For instance, certain invariant relationships decreed by physical constraints (such as the rigidity of the table), or specified by the designer (as in a sliding window), might be formulated as persistent definitions.

¤*5 Definitive programming reviewed*

The aim of the "definitive programming project" is to formulate a general-purpose programming paradigm based upon abstractly representing agent actions using definitive systems. Different aspects of this programme are represented in the collaborative work of the principal author with each of the co-authors; these will briefly reviewed below. For brevity and convenience, we refer the interested reader elsewhere for detailed illustration of our approach as appropriate, and indicate the principal directions for further work. There are surely many problems to be resolved - and yet to be encountered - in such an ambitious programme as we have framed above. The object of this paper is to explain why we believe that our methods represent a promising novel approach to sofware design and specification.

The basic principles of definitive programming are easily explained. The problem of specifying a system within a particular application domain is first addressed by developing an "underlying algebra", comprising relevant data types and operators upon these types, to suit the application. In the spirit of traditional approaches, specifying these basic operators is a concern at a lower level of abstraction. In the context of our work, it may be reasonable for these operators to be themselves specified by applying definitive principles (c.f. the discussion of complex operators for interactive graphics in [13]), but at some point we shall appeal to fundamental concepts and operations to be taken for granted. Note that there will in general be a useful role for axiomatic relations between operators in the underlying algebra in our approach, though this has yet to be much exploited.

Having established an underlying algebra, we shall introduce variables that can represent values of the various types. Through this generalisation, we abandon the referentially transparent principles underlying purely declarative programming, and have to rely upon a judicious use of variable definition to support our state-based paradigm. Each state transition is expressed via a definitive system formulated as a family of variables whose values are either defined explicitly, or defined in terms of other variables within the system and constants using the operators of the underlying algebra (and appropriate user-defined derived operators). We can think of the underlying algebra as providing the basis of generic concepts within which our specification is to be developed, and the variables as capturing the specific features of the specification pertaining to our particular application. Much richer categories of abstraction become accessible through the use of variables [6]: it is possible to formulate generic families of definitions to represent a class of objects of a certain sort for instance, as in an object-oriented framework. Perhaps most important, variables can be used to identify component parts of a value of complex type, enabling rich associations between complex objects to be established.

References [7] and [12] describe ways in which the above principles can be applied to the problem of designing a room layout. For this purpose, we can choose an underlying algebra whose data types are points, lines and shapes consisting of sets of points and lines together with simple geometric operators such as specify the join of two points, or the midpoint of a line. These primitive concepts form the basis of the definitive notation for line-drawing DoNaLD. Naive representations of objects such as doors, tables, lamps and power points within a room can then be formulated in such a way that in principle a generic object (such as the class door, defined by a system of variables to represent characteristic components together with definitions to establish appropriate relationships between these) can be described. It is also possible to express relationships between objects, e.g. to represent a lamp placed upon a table and connected by a cable to a power point, and to simulate the effect of moving the table. In many respects, our approach gives effective support for incremental design - for instance, we can exploit implicit definition of variables to express incomplete information e.g. to establish the functional relationships between the components of a door before assigning a location to the hinges, and so in effect represent a door whose current location is quite insignificant in the design.

The above discussion indicates how definitions can be used to support design, but cannot be said to capture what we should expect of a definitive program. A definitive system articulates a particular agent action; a definitive program must describe the way in which many actions interact. There are two complementary ways in which these possible transitions can be organised: associating actions according to the sequential agents by whom they are performed, and associating actions according to the patterns in which they can occur concurrently in simulation. These two approaches are respectively represented in our research by the LSD notation for concurrent systems [9], and the abstract definitive machine (ADM) [14].

Both LSD and the ADM have in common a framework of guarded action, whereby a procedural action, such as the redefinition of an explicitly defined variable, is performed in a context established by a definitive system. In LSD, the guarded actions make up the protocol for a sequentially acting agent, and the guards express enabling conditions for action to be met by the agent's view of the external system. We can very easily express simple constraints upon action in the LSD framework (such as prevent a user attempting to open a drawer that is obstructed by a table), but can only capture a loose synchronisation of action without introducing additional information about the environment in which agents operate (c.f. [9]). In so far as actions are organised by agent, an LSD specification is oriented towards the design rather than the simulation viewpoint (c.f. ¤4).

For purposes of simulation, we require a machine model that can represent synchronised action effectively, and that enables us to associate families of actions that occur concurrently in each system state. The ADM is an appropriate model within which to capture this multi-agent view. Within the ADM, the current system state is represented by the current values of a definitive system of variables, and the potential actions to be performed in the current state are represented by a system of guarded actions. Sets of definitions and actions that persist over the same period of time (e.g. such as are associated with the presence of a particular agent or object in the room) are grouped into entities. An ADM program takes the form of a set of abstractly specified entities to be dynamically instantiated and removed during program execution. Each action takes the form of a sequence of redefinitions or invocations / deletions of entities. In each cycle of the ADM execution all actions whose guards are currently enabled are performed concurrently (c.f. [14] for details).

To simulate the movement of objects about a room within the ADM, we shall need to introduce agents to maintain dynamically changing definitive systems that reflect the changing context for agent action. For instance, we shall require agents to impose constraints such as disjointness of physical objects in the room e.g. changing the context for action when two tables first touch (c.f. [4] ¤3 Principles #4, 5). Such simulation is beyond the range of our present prototypes, but we have been able to perform simple simulations with similar characteristics (see the block simulation in [11,12]). The ADM appears to have advantages as a machine model for concurrency [3,11], and in principle can support user resolution of conflicts such as arise when two agents attempt to move touching tables in opposite directions. Both LSD and the ADM represent an approach to modelling concurrency "from the study of relations and functional dependencies between events, rather than from the registration of absolute time intervals" [21], and promise to support a style of specification consistent with the proposals in [4].

LSD and the ADM are intended to provide the computational model of an intended behaviour for specification purposes. In so far as the primitive operations of the underlying algebra over which definitive systems are described can be implemented, our methods lead to executable specifications. For purposes of validation, it also remains to consider how the abstract computations we have described can be represented to the program designer. Neither the practical implementation nor the presentation is in general a trivial matter: when definitive principles are applied to the description of geometric modelling for instance, it is by no means obvious how conceptually simple operations such as forming an intersection of two complex implicitly defined objects should be implemented, nor how the resulting object should be displayed (c.f. [8]).

Much of our research has been devoted to these issues. In our framework, it is appropriate to regard the state of the screen from which the user obtains feedback as itself manipulated through definitive systems. That is to say, the current screen state is represented by a variable whose value is implicitly defined in terms of the internal system model over an appropriate underlying algebra. For simple graphical output, the underlying algebra of DoNaLD will suffice; for other purposes, we have developed an underlying algebra whose principal data type is a window of text [10]. In a serious simulation of a room in use (c.f. [7,8]), we would anticipate using such tools in combination to allow the user to monitor the current state (e.g. displaying a message as and when the door is obstructed by an object). The EDEN software package has been introduced to support the rapid prototyping of definitive notations of this kind in a standard UNIX/C environment [7].

Conclusion

By considering the nature of the software specification process, we are led to seek a style of programming that is cognitively based and supports incremental and interactive program development. A central technical problem is developing an appropriate means to model states and transitions both in design and in simulation for validation. Consideration of problems of architectural design suggests the adoption of a consistent mode of representation for both kinds of state-transition.

In the light of our analysis, a programming style based upon representing states by systems of variable definitions and transitions by redefinitions offers good prospects for software specification. Our preliminary attempt to describe general-purpose computation in this style - illustrated in [10] - makes use of an unusual abstract parallel machine model. Our current research aims at generalising this model with a view to characterising "definitive programming" definitively.

References

1. Abelson, Sussman The Structure and Interpretation of Computer Programs MIT Press
2. Backus J Can programming be liberated from the von Neumann style? Turing Award Lecture 1977, CACM 21, 8 (August) 1978, 613-641
3. Baldwin D Why we can't program multiprocessors the way we're trying to do it now Computer Science Tech Rep 224, University of Rochester 1987
4. Balzer R, Goldman N Principles of Good Software Specification and Their Implications for Specification Languages Proc IEEE Conf on Spec of Reliable Software, p58-67
5. Bernus P, ten Hagen P J W, Veerkamp P J, Akman V, IDDL: A Language for IIICAD Systems, Intelligent CAD Systems II: Implementation Issues, Springer-Verlag 1989, 58-74
6. Beynon W M Definitive principles for interactive graphics NATO ASI Series F, Vol 40, Springer-Verlag 1988, 1083-1097
7. Beynon W M, Yung Y W Implementing a definitive notation for interactive graphics New Trends in Computer Graphics, Springer-Verlag 1988, 456-468
8. Beynon W M, Cartwright A J A definitive programming approach to the implementation of CAD software Intelligent CAD Sys's II: Implementation Issues, Springer-Verlag 1989, 126-45
9. Beynon W M, Norris M T, Slade M D Definitions for modelling and simulating concurrent systems Proc IASTED conference ASM'88, Acta Press 1988, 94-98
10. Beynon W M, Norris M T, Russ S B, Slade M D, Y P Yung, Y W Yung Software construction using definitions: an illustrative example CS RR#147, Univ of Warwick 1989
11. Beynon W M Definitive programming for parallelism CS RR#132, Univ of Warwick 1988
12. Beynon W M, Cohn A J Representing design knowledge in a definitive programming framework, University of Warwick, September 1988
13. Beynon W M Evaluating definitive principles for interactive graphics New Advances in Computer Graphics: Proc CGI'89, Springer-Verlag 1989, 291-303
14. Beynon W M, Slade M D, Yung Y W Parallel computation in definitive models Proc Conpar'88 to appear
15. Glaser H, Hankin C, Till D Principles of Functional Programming Prentice-Hall 1984
16. Hayes I (ed) Specification Case Studies Prentice-Hall Int Computer Science Series 1986
17. Hughes J Why functional programming matters PMG Report #16, Chalmers Univ of Tech & Univ of Goteburg, 1984

18. Laird-Johnson Mental Models CUP 1983
19. McDermot A critique of pure reason Comput Intell 3, 151-160, 1987
20. Sartout W, Balzer R On the inevitable intertwining of specification and implementation CACM, 25 (7) (July), 438-440
21. Turski W M, Maibaum T S E The Specification of Computer Programs International Computer Science Series, Addison-Wesley 1987