Research report 147

SOFTWARE CONSTRUCTION USING DEFINITIONS: AN ILLUSTRATIVE EXAMPLE

W M Beynon, M T Norris†, S B Russ, M D Slade, Y P Yung, Y W Yung

(RR147)

We present and illustrate an approach to software construction that aims to develop a program through building an explicit state-transition model of the requirement. The approach is characterised by the use of systems of definitions that admit interpretation in both conceptual and machine-oriented terms. Such definitions represent a far-reaching generalisation of spreadsheet formulae that is well-adapted for interactive and incremental modelling. The method of elaborating the model of requirements is cognitively based. It depends upon the identification of agents and their privileges for action, as specified in the definition-based notation LSD, and the simulation of agent action in appropriate contexts within the Abstract Definitive Machine. This machine model has several interesting characteristics, including the explicit modelling of data dependency and the integration of data and process information, that supports new methods of addressing interaction and concurrency. We illustrate our method with reference to the development of a small educational program.

Keywords: Design methodologies, design representation, interactive software, modelling requirements, rapid prototyping, requirement specification, requirement analysis, software development.

Department of Computer Science
University of Warwick
Coventry CV4 7AL
United Kingdom

†British Telecom Research Centre
Martlesham Heath
Ipswich
United Kingdom

September 1989

Software Construction Using Definitions : an Illustrative Example

W.M. Beynon, M.T. Norris[1], S.B. Russ, M.D. Slade, Y.P. Yung, Y.W. Yung

Department of Computer Science, University of Warwick, Coventry CV4 7AL, UK
[1]British Telecom Research Laboratories, Martlesham Heath, Ipswich, UK

§1 Introduction

Over the past decade much work has been devoted to the problem of efficiently constructing and maintaining programs which animate an evolving requirement specification articulated in application-oriented terms. Recent surveys of the techniques being employed (or being proposed) offer a huge variety of methods with little consensus emerging on common features likely to be of lasting value for the future (see, for example [11],[13]). Rapid developments in the technology of distributed systems, concurrent processing, sophisticated interfaces and input devices, together with demand for ever larger and more complex software systems render the problem all the more urgent.

The purpose of this paper is to illustrate the application of a new approach to the design and development of interactive software. This approach - characterised by the use of systems of definitions - has already been used with considerable success in graphics [8][12][7]. Experience with graphical applications suggests that the methods and tools developed so far may have wide application to the modelling of complex requirements in a way which is at once intelligible and programmable. They may even serve to unify to some extent the current plethora of competing development methods and programming paradigms .

In order to demonstrate the scope and power of our definition-based methods it is sufficient to consider a surprisingly simple example. The idea for this example is taken from a small educational program called 'Jugs' available for several microcomputers used in UK schools [2]. The user is presented with a display of two jugs with integral capacities and the task is to achieve a specified target content of liquid in one of the jugs by filling, emptying and pouring between the jugs. The program animates these actions graphically in response to menu choices by the user and it can be used to investigate or illustrate simple number-theoretic principles.

As with much educational software we seek a specification of such a program which will allow implementation in a variety of languages, operating systems and hardware facilities. In the design of the program we should like to be able to experiment with effective screen layout and animation techniques; we might wish to distinguish use by teacher and pupil, or allow input by several pupils at once, or be able to customise the program for use by disabled pupils via special input devices etc. The obvious need here for rapid prototyping and for detailed modelling of the system of computer and

users is typical of numerous application areas. The simplicity of our example serves to illustrate our approach without misrepresenting many of the difficult issues surrounding the modelling of requirements. The design tools which are described here are themselves highly interactive and exemplify the belief that a useful requirements specification can only be built up incrementally with refinements and modifications (or 'elaborations' as referred to in [9]) being added as the details and complexities of a problem are realised.

In the course of the paper we shall illustrate the use of systems of definitions (and the associated tools we have developed for addressing issues of concurrency and synchronisation) by reference to the development of specific systems for modelling the construction and performance of a prototype Jugs program.

The use of application-oriented state-transition models is favoured as a means of describing parts of the requirement, for example in describing the state of a generic object in object-oriented design. It is sometimes tacitly assumed that global information about system state can only be conceived in machine oriented terms . For example [10, p.32]: "The state of a system is a list of the values held in all the memory of the system. States are an extremely implementation-dependent idea, so they should not appear in a requirements specification." The expressive power of a system of definitions stems from a dual interpretation: it can represent application-oriented transitions at a conceptual level, and explicit values suitable for machine interpretation.

We believe that this synthesis of conceptual and machine-oriented views can be used to construct a global state-transition model of the requirement that is both intelligible to the programmer and executable by the machine. In constructing such models we appeal directly to cognitive features of the application in order to identify not only appropriate variables but also the relevant agents and the ways in which they can act to change state. In the Jugs context for example we may wish to represent different privileges for the teacher and the pupil.

An important characteristic of our approach is that the elaboration of a specification exploits the same mechanism for representing change of state that is used in describing the automatic computation to be carried out using the computer. The use of similar techniques to integrate graphical modelling and animation is described in [12]. This integration enables the programmer to interact very closely with the computer during program development, and to introduce automatic techniques to support design within a single paradigm. In this way the programmer who anticipates changes to the requirement can make automatic provision for consistent change. While this uniformity of method renders the approach powerful and flexible, it also blurs traditional distinctions such as that between requirement specification and program specification, and between the design process and simulation.

The three principal sections of the paper outline and illustrate the main components of our state-transition modeller. §2 describes the definition-based methods we use to represent state, §3 makes

use of the notation (LSD) we have developed to model agents and their privileges to change state, and §4 illustrates the abstract machine model within which we specify programs to carry out automatic computation through transition from state to state.

§2 Fundamental principles

The principles behind our approach to program development will first be informally described. We focus on the abstract conceptual state-transitions that occur in the interaction between the computer and its environment: in the case of the Jugs program, the interaction between the pupil and the computer. For instance, suppose that one jug is full, the other empty and that the computer is awaiting an input from the user, as in Figure 1. Depending upon which input value is supplied i.e. which menu option is selected, the computer will enter a different state. If this option is appropriate, the computer will then carry out a sequence of transitions, in each of which it simulates pouring one unit of liquid until the requested action is completed. Throughout this process, the state of the screen display precisely reflects the current content of the jugs.

It will be convenient to refer to the state-transition model of the requirement to which we implicitly appeal in developing a program as the *conceptual* model. From the small sequence of transitions we have traced above, it is clear that even for a simple program the conceptual model is quite complex. Some idea of its complexity can be inferred from the fragment depicted in Figure 1. The significant point to observe is that if we develop a program that meets our requirements, it will serve as a computer representation of the conceptual model. What is more, if we can develop such a representation by any means whatever, it can readily be interpreted as a program.

Of course it is simplistic to refer to "the conceptual model of the requirement" as if it were easy to describe. The requirement is often elaborated in an incremental fashion and may be subject to revision retrospectively. For instance, we have not considered what the implications of allowing the user to supply input whilst the program is updating should be. It may also have some singularities: the consequences of the user simultaneously selecting two menu options may be unspecified. This motivates an incremental method of constructing a complex and possibly partial state-transition model with reference to the requirement.

In the state-transition models we construct, the method of representing state resembles that used in a spreadsheet: a state is represented by means of a system of definitions, each of which defines the value of a variable (cf a cell of the spreadsheet) either explicitly, or implicitly in terms of other variables and constants. Transitions from state to state are performed by redefining variables. The concept of state invoked here is not to be confused with the state of the spreadsheet display as reflected by the current values of variables alone. The definitions of the variables are significant in determining state, as is apparent from the fact that the same redefinition (or sequence of redefinitions)

has different effects depending upon the underlying relationships as currently defined. Similar considerations indicate it is not essential that the current values of variables can be determined. The significance of these observations will be clarified by illustration.

Consider how definitions can be used, as in [8][12], to specify transformations of a figure for graphical modelling and animation. If we are sufficiently ingenious, we can formulate a single system of definitions to serve several different functions. Certain definitions can capture both generic characteristics of a figure, such as invariant relationships between components, and specific characteristics of the particular figure imposed by the animator. Others may link the location of the figure to that of key reference points in such a way that a redefinition of the reference points describes a possible movement. Definitions may be viewed as providing a handle with which to modify the figure; possible transformations are identified by what variables can be redefined. The constraints upon such redefinition will be expressed in terms of agent protocols. It is significant that some of the definitions (e.g. specific characteristics) may be fixed when the animator's task is completed, but modified in the course of the design process. As remarked in [8], our representation of state and transition integrates modelling and animation: the same parameters used to change the length of Pinnochio's nose in animation also serve to specify it by design. When developing the state-transition conceptual model, this may be interpreted as bringing unifying principles to bear upon design and simulation (cf [4]). Such unification is the basis of powerful methods for addressing changing requirements.

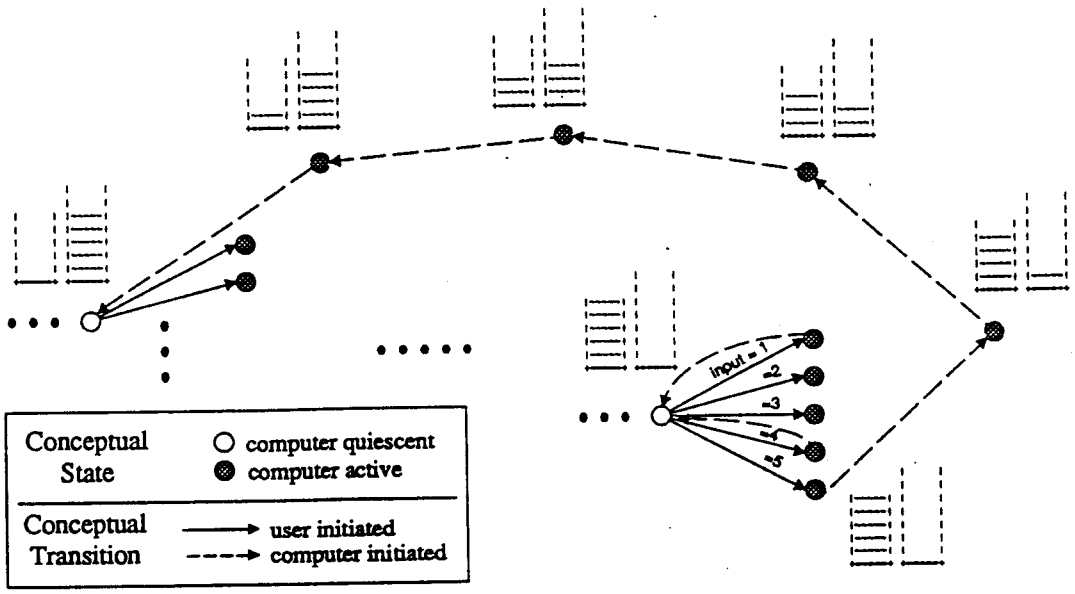


Figure 1: Tracing the selection and execution of the 'pour' option

To apply similar modelling techniques to the Jugs program, we first construct systems of definitions that serve to represent relevant design transformations or simulation transitions. The primary state information can be recorded by variables *capA*, *capB*, *contA*, *contB* to represent the capacities and contents of the jugs. The state of the screen display will at all times faithfully reflect the values of these variables. For instance both the depiction of jug A and the status of the menu options will be determined by their current values. Formally, the screen display will be represented by a variable, whose value is functionally defined in terms of the core variables. For this purpose we introduce data types and operators to describe the screen layout. The choice of these data types will be determined by the nature of the display interface. Outline details of such a specification appear in Figures 2 and 3.

capA = 7 *// these variables need not be defined explicitly*
contA = 5
fullA = (capA == contA)
emptyA = (contA == 0)
.....
pupilmenu = [("Fill A" , $avail_1$) , ("Fill B" , $avail_2$) ,
("Empty A" , $avail_3$) , ("Empty B" , $avail_4$) , ("Pour" , $avail_5$)]
realmenu = pupilmenu ++ [("Pour A to B", $avail_6$) , ("Pour B to A", $avail_7$) ,]
$avail_1$ = ¬fullA
$avail_2$ = ¬fullB
$avail_3$ = ¬emptyA
$avail_4$ = ¬emptyB
$avail_5$ = $avail_6$ ∨ $avail_7$
$avail_6$ = (¬emptyA) ∧ (¬fullB)
$avail_7$ = (¬fullA) ∧ (¬emptyB)
.....

Figure 2: Primary state information in the Jugs program

The description of the screen layout in Figure 3 is expressed in the definitive notation SCOUT. The data types in SCOUT are:

integer	e.g. n, capA
point = integer × integer	e.g. base
box = point × integer × integer	e.g. contAbox
frame = *list of* **box**	e.g. jugAbox
window = frame × string × attribute	e.g. capAwindow
display = *list of* **window**	e.g. screen

The values of SCOUT variables are associated with where, what and how information is to be displayed. If we wish to display the Jugs program output as mapped out in Figure 3a, we can formulate the relationships between the locations of each item via the system of definitions in Figure 3b. This gives the program designer a means to relocate display elements conveniently in a consistent fashion. The display is made up of windows, each of which is defined by a list of boxes to specify its shape, a string that declares its content, and an attribute field that specifies how the string is to be displayed. Dynamic relationships between the values of internal parameters and display characteristics are a powerful feature of SCOUT. Although *contA* and *availbutton*$_i$ are both integers, they are revealed in quite different ways. The water level, *contA*, is represented by a block of ~'s but *availbutton*$_i$ is reflected through the use of normal or inverse video in the menu display.

Several different roles for definitions are represented in Figures 2 and 3. In a conventional procedural implementation, the string fields in the menu definition would be embedded in the program text, the value of capA would typically be fixed throughout each execution (e.g. as determined by the teacher or through random generation), the value of contA would change dynamically throughout the simulation. Some definitions, such as that of fullA and emptyA, express semantic relationships which cannot reasonably be changed. Other definitions determine features of the screen display so that they can be conveniently modified to suit the program designer's preference.

We view the process of program development as incrementally developing a state-transition model of the requirement. In constructing this state-transition model, we shall exploit the fact that there is a rich family of latent transitions - described by redefining one or more variable - associated with each system of definitions. By building up appropriate systems of definitions we can readily describe the kind of state-transitions that appear in the conceptual model. For instance, the definitions in Figures 2 and 3 guarantee that simply redefining contA to be the value of capA simulates transition to a state in which jug A is full - all the changes associated with redefining contA (e.g. the new status of the flag fullA and the menu option "Fill A", the modifications to the display) are implicit. In this way, a complex transition is modelled in a conceptually simple manner. In passing, it may be noted that the complex synchronised update of state that a definitive model describes is encountered in real applications, where e.g. the act of signing a document effects a significant contingent change *by definition.*

§3 Agents and protocols

Systems of definitions provide the basic state representation needed for developing the conceptual state-transition model. Though a system of definitions such as Figure 2 can describe a consistent state, it does not give any information about acceptable transitions. There is no context in which redefining fullA in Figure 1 as "capA==capB" is appropriate for instance. Our state-transition models are constructed in two phases, both of which have a cognitive aspect. In the first phase, we derive a partial specification of a state-transition model by considering what agents can change the state of

system and when and how they are privileged to act. This is sufficient to constrain the possible transitions considerably - it precludes redefining fullA for instance - but does not take full account of assumptions about the environment in which agents act. In the second phase, as described in §4, we refine our model to this end.

By way of illustration, in the execution of the Jugs program, different possible state transitions from a given state can be carried out by the teacher, the pupil or the computer when appropriate enabling conditions for action are met. The pupil is conditionally permitted to select menu options to initiate the pouring of water into, out of, and between the jugs, whereas the teacher can instigate a wider range of possible actions, such as changing the capacity of a jug. The computer acts as an agent through automatic computation invoked by the selection of a menu option.

The interaction between the pupil and the computer in the Jugs program illustrates agent privileges, but is simple enough to analyse informally. It may be helpful to refer at this point to the definitive version of the Jugs program in Figure 4 below: this illustrates how the redefinition of variables closely reflects the privileges of agents acting in the application. In general, a more systematic approach to the identification of agent privileges is required.

We introduce the language LSD as a medium for describing the capabilities and perceptions of agents within a system (cf [5]). By means of an LSD specification, we shall describe the conceptual model partially, identifying possible actions by agents (represented by redefinition) in states (represented by definitive systems). In this connection, the possibility of concurrent action by several agents, represented by the simultaneous redefinition of (independent) variables within a system of definitions, is of particular interest. To illustrate our methods effectively, we shall discuss LSD with reference to a specification that addresses the dialogue_manager(), the pupil() and the computer as concurrently acting agents (see §4). Such a specification is appropriate, for example, if we intend to implement a version of the Jugs program that makes use of an unusual interface for menu selection, such as a set of mechanical switches that can be depressed and released at will.

The basic concepts of LSD will be described with reference to the outline LSD specification in Figure 4. The primary concept is the *agent*. For each agent we distinguish three kinds of variable: **oracle** variables that represent values that influence the behaviour of the agent and that are subject to change beyond the agent's control, **state** variables that are conditionally under the control of the agent, and **derivates** recording relationships that invariably form part of the context for the agent's actions. Each variable is bound to a particular agent, but may be referenced as a state, oracle or even as a derivate variable by other agents. The variables bound to an agent are indicated by a '#' prefix. Agents are presumed to act sequentially in accordance with a protocol. The actions in a protocol take the form of guarded sequences of redefinitions or invocations of agents: if a guard is true in a given state, the corresponding sequence of actions is enabled. Note that one agent can act directly to change the value of a variable bound to another, in contrast with an object-oriented paradigm.

An LSD specification is interpreted in state-transition terms by conceiving a system of sequential agents acting asynchronously. The current state is determined by what agents are instantiated and what action(s) are currently enabled. The state is represented by the system of definitions of variables bound to these agents. Possible transitions from the current state are associated with the parallel execution of a non-empty subset of the enabled actions. A transition may have the effect of changing the set of instantiated agents, as agents are created or destroyed. It may also serve to introduce temporary redefinitions of a variable, such as the definition of the contA in terms of contB that guarantees that changes in the content of the jugs are synchronised when pouring (cf pour(5)). The deletion of an agent can also be associated with the setting of a special variable LIVE to false (cf pour() in Figure 4).

An outline LSD specification of some of the key agents acting when a pupil is using the Jugs program appears in Figure 4. Agents such as pupil(), jugA() and screen() are permanently instantiated. The agent pour(i) is instantiated only intermittently, when a menu option has been selected. The dialogue_manager(), screen() and pour() agents capture different aspects of the activity of the computer that are typically carried out by independent processors. These agents may be respectively associated with the detection of mouse activity (for menu selection), with display processing (for screen update and refresh) and with computation in the central processor (execution of the updating procedures).

With discretion, an LSD specification can be interpreted in operational terms, but only by admitting the possibility of singular behaviour in certain circumstances. In view of the singular nature of the conceptual model itself, this is only to be expected. The parallel execution of enabled actions may genuinely lead to conflict, as when two people attempt to pick up the same object simultaneously. (The loose analogy with functions with singularities in real analysis may be helpful here.) There is also a danger of reading too much behavioural information into a specification. LSD agents are not obliged to act at any particular speed, so that synchronisation cannot be guaranteed. In the context of Figure 4, it is sufficient to assume a shared variable model of communication, but there are circumstances when communication is asynchronous. In these cases, a variable v that serves as a state variable for one agent and an oracle for another may have to be represented by two independent variables v.s and v.o. Our approach to refining the state-transition model supplied by a LSD specification to address such issues will be described in more detail below.

An LSD specification provides a subtle model of agent privileges and protocols, as will be illustrated with reference to Figure 4. As described in its protocol, the actions open to the pupil agent consist of selecting one of the available menu options. For the pupil, $availbutton_n$ is sensory input. The screen agent ensures that the variable availbutton faithfully reflects the status of the appropriate menu option e.g. by using different colours in the display. The protocol of the pupil may be interpreted as representing intelligent behaviour on the part of the pupil, who does not attempt to select invalid options. If we wish to adopt protocols of this nature, we must ensure that the oracles

required by the agent correspond to perceived values. Such an interpretation might be inappropriate for a colour-blind pupil, for instance.

An alternative, less plausible interpretation of the pupil protocol is that a button is disabled when the associated menu option is invalid. Without this assumption, provision must be made for the irresponsible pupil who is prepared to press any button at any time, according to the protocol:

$$\text{TRUE} \rightarrow \text{button}_i = \text{TRUE} ; \text{button}_i = \text{FALSE}.$$

If this amended protocol is substituted in the LSD specification in Figure 4, an invalid menu selection elicits no visible response; pour(i) is a null procedure in this case. Alternative responses are readily described. The protocol of the dialogue_manager() might be amended so that pour(i) is invoked only if avail_i. Feedback to the user could be given by introducing the derivate:

$$\#\text{error} = \text{button}_i \wedge \neg\text{avail}_i$$

into the specification of the screen() agent. These examples illustrate the direct correspondence between a cognitive model and the agent-oriented LSD specification.

agent jugA() {
oracle #contA = 0, #capA = 7
derivate
    #emptyA = (contA == 0),
    #fullA = (contA == capA)
}

agent pour (i) {
state #input = i
derivate #LIVE = updating
state contA, contB
protocol

$(\text{input} == 1) \wedge \text{avail}_1 \rightarrow \text{contA} = \lvert \text{contA} \rvert + 1$,		// Fill A
$(\text{input} == 2) \wedge \text{avail}_2 \rightarrow \text{contB} = \lvert \text{contB} \rvert + 1$,		// Fill B
$(\text{input} == 3) \wedge \text{avail}_3 \rightarrow \text{contA} = \lvert \text{contA} \rvert - 1$,		// Empty A
$(\text{input} == 4) \wedge \text{avail}_4 \rightarrow \text{contA} = \lvert \text{contA} \rvert - 1$,		// Empty B
$(\text{input} == 5) \wedge \text{avail}_j \rightarrow \text{contB} = \lvert \text{contA} + \text{contB} \rvert - \text{contA}$; input =j,	(j=6,7)	// Pour
$(\text{input} == 6) \wedge \text{avail}_6 \rightarrow \text{contA} = \lvert \text{contA} \rvert - 1$,		// Pour A to B
$(\text{input} == 7) \wedge \text{avail}_7 \rightarrow \text{contA} = \lvert \text{contA} \rvert + 1$,		// Pour B to A
$\neg\text{button}_{\min(\text{input},5)} \wedge \neg\text{avail}_{\text{input}} \rightarrow \text{updating} = \text{FALSE}$		

}

agent dialogue_manager() {
oracle #button$_i$
state-oracle #updating = FALSE
protocol
    ¬updating ∧ button$_i$ → updating = TRUE ; pour(i) (1≤i≤5)
}

agent screen() {
derivate
    #availbutton$_i$ = avail$_i$ ∧ ¬updating (1≤i≤5)
}

agent pupil()
oracle availbutton$_i$ (1≤i≤5)
state button$_i$ (1≤i≤5)
protocol
    availbutton$_i$ → button$_i$ = TRUE ; button$_i$ = FALSE (1≤i≤5)
}

jugA()
jugB()
dialogue_manager()
screen()
pupil()
.....

Figure 4: An outline LSD specification for the user input protocol

§4 Simulation and Control

In the first instance an LSD specification is to be interpreted as expressing the privileges of agents. That is, it does not specify when an agents acts, merely the enabling conditions for actions to be performed [5]. For example, *if* a button is depressed and updating is not in progress, the dialogue manager agent is *permitted* to set the updating flag and proceed to invoke the pour agent. This style of specification is consistent with our objective of describing a state-transition conceptual model, since it is a means of expressing transitions that are possible in each state, but it does not take full account of the context in which interaction occurs. A state-transition model is executable only in the restricted sense that its behaviour can be simulated subject to supplying or generating appropriate inputs. In this section, we shall describe an abstract machine within which the state-transition model supplied by an LSD specification can be executed. What is more significant for practical purposes is that we can refine an LSD specification into a program that reflects assumptions about communication

channels, response times, execution speeds and synchronisation. This means for instance that the interface between the dialogue_manager(), the pupil() and the computer specified in Figure 4 can be studied through simulation under various conditions.

The issues can be illustrated with reference to Figure 4. Some operational characteristics of the Jugs interface are captured in the LSD specification. Because an LSD agent acts sequentially, at most one button will be pressed at any time. Whilst the updating flag is set, no available menu options will be displayed to the pupil(). If we presume that pushing a button is a matter of intelligent choice, rather than an obligation upon the pupil, there are still constraints on the response to button pushes. Once the dialogue_manager() has set the updating flag, it is inhibited from responding to a depressed button until the updating action has been completed. There are other synchronisation issues to consider however.

In using the Jugs program, the pupil() agent acts from time to time to select an available button. The operation of this button may resemble a double-click of a mouse, so that the time interval during which $button_i$ is true is constrained. It might on the other hand be a mechanically operated switch, than can be pushed and released at will. It is assumed that the button depression is detected and acted on by the computer. The intended operation of the dialogue_manager() agent involves detecting when a button is depressed and, if the context is appropriate, performing the requested operation by invoking the pour() agent. Clearly, the speed of execution of the dialogue_manager() agent should exceed that of the user() agent, since otherwise some button pushes may be ignored. The relative execution rates of agents represents part of the intended semantics of the model which cannot easily be captured within the agent specification.

There are indeed simulations in which agent actions consistent with the LSD protocol lead to behaviour that is hard to foresee. Consider for instance the following scenario: suppose that the buttons are mechanical switches that can be depressed if and only if the corresponding menu option is deemed to be available (a "fail-safe" system in which the selection of inappropriate menu options is apparently physically guaranteed). Suppose that jug B is full and jug A empty. Suppose that the pupil() attempts to select the menu options "Fill A" and "Pour" by pushing buttons 1 and 5 in rapid succession, and holding down button 5. The dialogue manager may detect the depression of button 1, commit to the "Fill A" action according to its protocol, but not set the updating flag until after button 5 has been depressed. When jug A has been filled, the dialogue_manager will then - according to its protocol - be allowed to act as if the now invalid "Pour" option had been invoked.

Such scenarios can be simulated using the *abstract definitive machine (ADM)*. A full discussion of the ADM appears in [6]. Within the ADM, state information is recorded using definitive systems, but there is also provision for automatic changes of state according to a prescribed program. An ADM program is expressed by specifying *entities*, each of which comprises a set of definitions and a set of actions. In execution, the behaviour of the program is determined by the set of entities that is currently

instantiated; this set can change dynamically as actions invoking or deleting entities are performed. The current state of the executing ADM program is determined by the system of extant definitions i.e. definitions associated with currently instantiated entities. The transition from state to state is prescribed by the system of extant actions. Each action takes the form of a guard (a boolean condition expressed in terms of variables whose definitions are extant) together with a sequence of instructions that redefine variables or invoke or delete entities.

The ADM operates in a synchronous fashion. On each machine cycle the guards of all extant actions are evaluated in the context supplied by the extant definitions and those sequences of actions associated with true guards are performed in parallel. Parallel redefinition can lead to conflict, as when the definitions "a=b" and "b=c; c=a" are to be executed simultaneously. The unusual qualities of the ADM include the graceful detection of such conflicts and of exceptional conditions such as arise when the value of an undefined variable has to be supplied. Because of the nature of the computational model, it is easy for the programmer to intervene to resolve conflict or to supply input (cf user_input in Figure 5).

By way of illustration, an ADM program that can be used in conjunction with the systems of definitions in Figures 2 and 3 to construct a prototype Jugs program is given in Figure 5. The pupil supplies input when the ADM invokes init_pour(user_input) and user_input is undefined. (The expression |contA| designates the current value of the variable contA.) The close resemblance between the sequence of transitions performed by the ADM in execution and the conceptual model of requirement of the Jugs program has already been remarked (cf Figure 1).

entity pour (option)
action (option == 1) $\wedge$ $avail_1$ $\rightarrow$ contA = |contA| + 1 ;
(option == 2) $\wedge$ $avail_2$ $\rightarrow$ contB = |contB| + 1 ;
(option == 3) $\wedge$ $avail_3$ $\rightarrow$ contA = |contB| - 1 ;
(option == 4) $\wedge$ $avail_4$ $\rightarrow$ contB = |contB| - 1 ;
(option == 6) $\wedge$ $avail_6$ $\rightarrow$ contA = |contA| - 1 ;
(option == 7) $\wedge$ $avail_7$ $\rightarrow$ contA = |contA| + 1 ;
(option == i) $\wedge$ $\neg avail_i$ $\rightarrow$ updating = FALSE ; delete pour(option) ; (1≤i≤5)
(option == 6) $\wedge$ $\neg avail_6$ $\rightarrow$ updating = FALSE ; contB = |contB| ; delete pour(option) ;
(option == 7) $\wedge$ $\neg avail_7$ $\rightarrow$ updating = FALSE ; contB = |contB| ; delete pour(option) ;
}

entity pupil () {
action
$\neg$updating $\rightarrow$ init_pour(user_input) ; updating = TRUE ;
}

```
entity init_pour (input) {
owner: option = input ;
action    (option == 5) ∧ avail6 → contB = |contA + contB| - contA ; option = 6 ;
          (option == 5) ∧ avail7 → contB = |contA + contB| - contA ; option = 7 ;
          (option ≠ 5) ∨ ¬avail5 → pour(option) ; delete init_pour(input) ;
}
```

Figure 5: The Jugs program for the Abstract Definitive Machine

§5. Conclusion

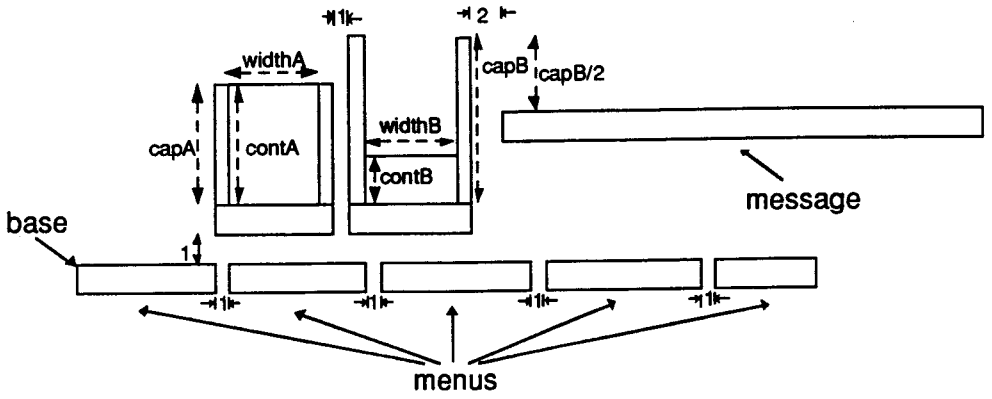
In this paper, we have illustrated a definitive ("definition-based") style of programming that is constructed around explicit state-transition models in which states are represented by systems of definitions and transitions by redefinition. The illustrative programs were developed using working prototypes of the Abstract Definitive Machine and the SCOUT interpreter. The design and implementation of these systems has been closely associated with the development of the EDEN interpreter, a UNIX/C program that exploits a mixed definitive/procedural programming paradigm. The significant role of EDEN reflects the fact that the adoption of a pure definitive programming style would require the development of many special purpose definitive notations (such as SCOUT or DoNaLD [7]), if not the design of special-purpose hardware to justify the treatment of the SCOUT screen (Figure 3) as a genuine definitive variable. This means that in practice it is convenient to support systems of definitions over a sufficiently expressive underlying algebra (list-based in EDEN) and invoke procedural methods to handle system-oriented processing. A detailed description of these implementation methods appears in [7]. With the special problems of portability of educational software in mind, we are currently investigating the prospects for translating from definitive programs into a procedural form. It is significant that this translation process requires prior knowledge of what variables are subject to redefinition; a further indication of the close relationship between program design and program execution in a definitive approach.

Despite the limitations of our present prototypes, experience with definitive programming systems confirms the benefits of using definitions in program design and prototyping. The potential of SCOUT as a means of specifying and manipulating display layout is clear. Our experiments with the ADM demonstrate the relative ease with which the programmer can interpret a suspended computation and intervene to redirect the computation as required.

In the longer term, our methods point towards a mode of programming in which the formal expression of the requirement takes the form of an application-oriented state-transition model that can be directly interpreted by the computer. If such a vision can be realised, it will bring about a much closer correspondence between requirement and program than exists at present, establishing a direct

link between the incremental development of a program and the formal analysis and representation of the requirements.

There are of course many technical problems to be overcome in the development and exploitation of definitive methods for large applications [3]. Our research hitherto has focussed upon the distinctive characteristics of a definitive programming style as illustrated by the ADM: automatic support for data dependency, the integration of data and control information and good prospects for concurrent implementation [6]. At the same time, there are significant connections with the major programming paradigms to be exploited. Functional programming languages such as ML, for instance, incorporate techniques for specifying complex data types and operators that provide the ideal basis for an expressive and extensible underlying algebra; logic programming techniques enable the translation of equational constraints into systems of functional relationships resembling systems of definitions; object-oriented methods provide sophisticated generic models of objects that can serve as pre-programmed components of the definitive models illustrated above. When integrated with such existing approaches, it is to be hoped that definitive methods can make a significant contribution towards realising the objectives for future programming language design set out by Backus in [1].



(a)

```
base = {1.c, n.r}          # 1 char-width(.c) right and n char-height(.r) down from origin
menu1box = [base, 1, strlen(pupilmenu1)]
        # a box whose NE corner is at base, 1 char-height and strlen(menu1) char-width
menu2box = [menu1Box.ne + {1.c, 0}, 1, strlen(pupilmenu2)]
jugAboxes = ([menu1box.ne+(0, –(2+capA).r}, capA, 1],
             [menu1box.ne+{(widthA+1).c, -(2+capA).r}, capA, 1],
             [menu1box.ne+{0, -2.r}, 1, widthA+2])
jugBboxes = ([jugAboxes.2.sw + {2.c, -capB.r-1}, capB, 1], ...
contAbox = [jugAboxes.1.sw+{1.c, -contA.r}, contA, widthA]
messagebox = [jugBboxes.2.ne +{2.c, (capB/2).r}, 1, strlen(status)]
...
```

(b)

```
background_i = availbutton_i ? BG_BLACK : BG_WHITE
                                          #reverse background  if option invalid
cA = repeatChar('~', widthA*contA)        #use '~'s to represent water level
jugA = repeatChar('|', 2*capA)//"+"//repeatChar('=', widthA)//"+"
...
capAwindow = <jugAboxes, jugA, BG_WHITE>
    # form a window by putting string jugA (what to display) into boxes of jugAboxes
    # (where to display) with attribute BG_WHITE (how to display)
contAwindow = <(contAbox), cA, BG_WHITE>
...
screen = ( menu1window > menu2window > ... > contAwindow > capAwindow > ... )
    # screen represents  the physical screen; it displays the windows listed.
    # If windows overlap, menu1window overlays menu2window etc.
```

(c)

```
                |      |
                |      |
|~~~~~|         |      |
|~~~~~|         |      |   Target is 1 : awaiting input
|~~~~~|         |      |
|~~~~~|         |      |
|~~~~~|         |      |
+=====+         +======+

1:Fill A 2:Fill B 3:Empty A 4:Empty B 5:Pour
```

(d)

Figure 3: (a) Screen layout (b) definitions for locations (c) other SCOUT definitions (d) Sample output

§6 References

[1] J Backus *Can programming be liberated from the Von Neumann style?* CACM **21**(8), pp.613-641, 1978

[2] Original version written by Ruth Townsend - the version for the BBC which inspired our example is distributed by the Chiltern Advisory Unit.

[3] W M Beynon, A J Cartwright *A definitive programming approach to the implementation of CAD software* to appear in Intelligent CAD: Implementation Issues, Springer-Verlag 1989

[4] W M Beynon *Definitions as a framework for design* Proc 3rd Eurographics Workshop on Intelligent CAD, CWI Amsterdam, 1989

[5] W M Beynon, M Norris, M Slade *Definitions for modelling and simulating concurrent systems* in Applied Simulation and Modelling '88, Acta Press, 1988

[6] W M Beynon, M Slade, Y W Yung *Parallel computation in definitive models* to appear in proc. Conpar 88, UMIST, Sep. 1988

[7] W M Beynon, Y W Yung *Implementing a definitive notation for interactive graphics* New Trends in Computer Graphics, Springer-Verlag, 1988

[8] M Chmilar, B Wyvill *A Software Architecture for Integrated Modelling and Animation* New Advances in Computer Graphics, Proc. of CG I '89, pp.257-276

[9] M S Feather *Constructing Specifications by Combining Parallel Elaborations* IEEE Transactions on Software Engineering **15** (2), 198 - 208, 1989

[10] D A Lamb *Software Engineering: Planning for Change* Prentice-Hall Int'l, 1988

[11] B Ratcliff *Software Specification and Design* in "Software Engineering: Principles and Methods", Blackwell 1987

[12] B Wyvill *An interactive graphics language* PhD thesis, University of Bradford, 1975

[13] S Y Yau, J J-P Tsai *A Survey of Software Design Techniques* IEEE Transactions on Software Engineering **12** (6)., pp.713 - 721, 1986