A Computer-Aided Script Generator for Computer-Aided Design

Yun Pui Yung, Meurig Beynon
Department of Computer Science
University of Warwick, Coventry CV4 7AL, UK

Abstract

Integration is a key issue in Computer-Aided Design. It should be possible to combine the information generated by the different designers in a convenient manner. In a recent paper we have discussed a multi-agent computational model within which we propose to coordinate the activities of the design agents in a concurrent engineering design environment [ABCY94]. Our agent-oriented computational model is represented using definition-based (or *definitive*) scripts. This paper focuses on how information produced at the drafting stage can be integrated into the later stages of the design process. We describe a two-view computer-aided script generator that allows us to draft a drawing via a graphical user-interface and to edit its associated textual description as a definitive script. This generator is itself implemented in our multi-agent computational model.

1. Design and Definitive Script

The purpose of computer-aided design is to go beyond the computer-aided drafting stage, to produce useful information for manufacturing. CAD packages generally use graphical user-interfaces. The power of such systems in drafting derives from pre-programmed graphical components and procedures to manipulate them. For instance, many CAD packages provide libraries of templates of design components, consistency checking and auto-routing.

Design principles based on the selection and semi-automated assembly of pre-constructed components can be exploited beyond the drafting stage. In [EDC93], for instance, Sharpe reports work on a rule-based system to deal with entire design process, from conceptual design right through to manufacturing. His system includes a catalogue of design components with known properties. By interpreting specified rules for design. the computer generates a set of candidate designs from which the designer can build simulations and select the final design.

Our experience in using definitive scripts in design strongly suggests that a GUI has intrinsic limitations, however powerful the objects behind the interface. An intelligent CAD package should include facilities for the designer to express design decisions in other ways. In a previous paper, we have argued that support for the concurrent engineering process can be best enhanced by improving the computational model that we use [ABCY94]. We advocate the use of a multi-agent definition-based computational model. Our purpose in this paper is to explain how direct manipulation methods can be used in conjunction with such a model.

The design environments we develop are not characterised by sophisticated interfaces giving access to complex pre-programmed objects. The computational methods we exploit provide the user with no preconceived components, but with a powerful latent functionality that is well-adapted for the synthesis of complex objects from primitive structures. The scripts generated in the design process are not only a documentation of the design but also a *virtual prototype* that can be examined, used and altered by the designers involved.

The virtual prototype of a design is developed as a collection of definitive scripts together with specified agent privileges to observe and act upon them. The fundamental principle behind this agent-oriented definitive computational model is 'observation and experiment'. A definitive script is very similar to a generalised spreadsheet, in that it allows a user to perform "what if?" experiments. In a spreadsheet, the user can change the value of a cell and the value of the related cells will be automatically updated. In a definitive script, when a variable is redefined, the values of the variables dependent on the redefined variable will also be updated. This indivisibility of propagation of changes establishes a close association between the values of variables in a definitive script and observations made in the context of the experiment.

2. Two-View Editing of Scripts

Although our agent-oriented definitive computational model is a general framework for design, we will mainly be concerned in this paper with its application to CAD drawing. Within our design framework, the output of a CAD drawing tool should ideally be represented by

definitive scripts. This motivates us to consider a CAD drawing tool as a computer-aided script generator.

Although the generation of a script is our eventual objective, working directly with scripts is not always a pleasant experience. The graphical workstation has become the basic resource for CAD systems. It offers the advantage of enabling us to manipulate information in two-dimension. Both 2D, and to some extent, 3D drawings can be created, modified and presented more effectively using *direct manipulation* methods. However, the kind of information that can be generated through direct manipulation is restricted by the user-interface. Our aim is to create an editing environment that captures the advantages of direct manipulation whilst retaining the full power of script development. We compare direct manipulation and direct editing of scripts in more detail.

Functionality. Direct manipulations are preconceived methods of transforming objects encoded in a graphical user interface. Defining new operations on these objects is therefore out of the scope of direct manipulation. By editing scripts directly, it is possible to manipulate objects, to define new operations with existing objects, and even to define new primary shapes and associated operations. As a powerful illustration of this, the two-view editor we are developing is itself written entirely using the same kind of script as is used to represent the output drawings. We can modify the user-interface on the fly in just the same way that we can modify the drawing through editing the script form.

Accuracy. A graphical interface puts a constraint on the accuracy with which we can represent the scalar attributes (such as the dimensions) of objects. This inaccuracy can be attributed both to the sensitivity of the pointer device and to the resolution of the screen display. Scripts do not suffer from these problems because scalar values are represented by abstract numbers in a script rather than by visual metaphors.

Integrity. A user-interface can restrict the user to performing safe transformations of a design. Certain modifications of the underlying script can be prohibited by the interface. There is no way to guarantee that the integrity of a design is respected in a direct editing

environment. For instance, we can design an interface such that the relocation of any corner of a rectangle will translate the whole rectangle, whilst editing the location of a corner directly in the script may distort its shape.

Visualisation. Direct manipulation uses graphical metaphors for objects, whilst a script represents objects by strings of text. Since the effect of a change in the object is often easier to appreciate by representing the object graphically, direct manipulation provides a shorter feedback loop for faster iteration of design.

Semantics. Although, in direct manipulation, what you see looks like an object and what you can do with it may sometimes have a corresponding real-world meaning, it is hard to identify the drawing you are working on with a real-world object. The identity of an object is closely related to its intended usage. Consider representing a table by a rectangle. Moving the rectangle can be interpreted in real-world terms as moving the table. But it is difficult to find a real-world meaning for resizing the rectangle. In effect, the range of transformations of graphical objects supported by the system is restricted, so that the fact that certain transformations of a graphical image can be interpreted is coincidental. In contrast, the relationship between transformations of the graphical image and those of the real-world object it represents are explicitly derived from observation and experiment [BR92].

Both editing through GUI and directly editing the script are important in their own right. In the two-view document editor Lilac, for example, the user would input via the WYSIWYG view most of the time when dealing with general editing of text whilst using the script view to modify the formatting styles [Brooks91]. A similar activity is involved in computer-aided design, where we not only want a drawing of the design product, but a design that can be tested. We need more semantic information to complement the piece of drawing. We ideally need a two-view editing system in which we can construct a substantial drawing via a GUI and carry out more subtle manipulations through direct editing of the generated script. We believe that this would allow better integration of the editor output to the rest of the design.

3. Nature of the Scripts

To fulfil our expectations of a two-view computer-aided script generator, the underlying script that is generated should be such that:

- it can be efficiently interpreted as a drawing so that the two views can be closely synchronised.
- it can be easily adapted to be more than a drawing, to express its role in the virtual prototype.
- it is easy to integrate with other such scripts to form a coherent design.

Definitive scripts have these properties. In a definitive script, each definition associates a formula with a variable. This formula either specifies the value of the variable explicitly (as in $a=5$) or implicitly in terms of other variables (as in $a=2*b+c$). In operational terms, the presence of the definition $a = 2*b+c$ in a script means that if the value of b or c is updated, the value of a will be indivisibly maintained to the value $2*b+c$. Indivisible propagation of change is a key concept in the definitive representation of state. Through one redefinition, an arbitrarily large set of values of variables may be updated automatically. These values will always be consistent with the formulae associated with the variables. Whenever the value of a variable is referenced, the system will guarantee to return its most up-to-date value.

The application of definitive principles to interactive graphics was first introduced by Brian Wyvill [Wyvill75], who has since exploited them in graphical design and animation [CW89]. The formal concept of definitive notations for formulating definitive scripts was introduced by Beynon [Beynon89]. We have developed several definitive notations for interactive graphics. These notations have data types and operators that are suitable for describing graphical objects. We have a definitive notation for 2D line drawing DoNaLD, a 3D geometric modelling notation CADNO and others. A redefinition to a graphical object in one of these notations will cause an automatic indivisible updating of its image on the screen display.

Indivisible propagation of change is a property common to definitive scripts and spreadsheets. Both definitive scripts and spreadsheets are similar in that they minimise the effort of creating a user-interface. A spreadsheet incorporates a user-interface that makes the spreadsheet user the primary implementer of applications [JNZM93]. In a similar spirit, in definitive notations, the procedures for updating the image caused by a change of design are hidden. For example, redefining the variable representing the centre of a circle will automatically drive some procedures to redisplay the circle on the screen. In this way, the abstract design and its visualisation are closely coupled. This allows the designer to pay more attention to issues directly related to the design rather than to peripheral technical issues concerned with visualisation.

Several definitive notations can be used in combination. This is well-illustrated by our prototype of the two-view script generator. Definitions in each definitive notation are like a description of a specific perspective of design. In our script generator as depicted in Figure 1, DoNaLD is used for specifying the icons of the buttons whilst another definitive notation, SCOUT, is used for describing the arrangement of the window layout. The SCOUT and DoNaLD notations are interpreted in a special-purpose Evaluator for Definitive Notations EDEN. EDEN is also a definitive notation with ordinary arithmetic operators that supports the specification of abstract models (e.g. engineering models defined by laws of motion [BBY92]). The graphical definitive scripts hence serves as a visual interface to the abstract model. The nature of definitions enables a total separation of the development between the abstract model and its visualisation. This visualisation mechanism has properties analogous to a mechanical linkage [BY90]. This makes definitive scripts very effective for combining design fragments.

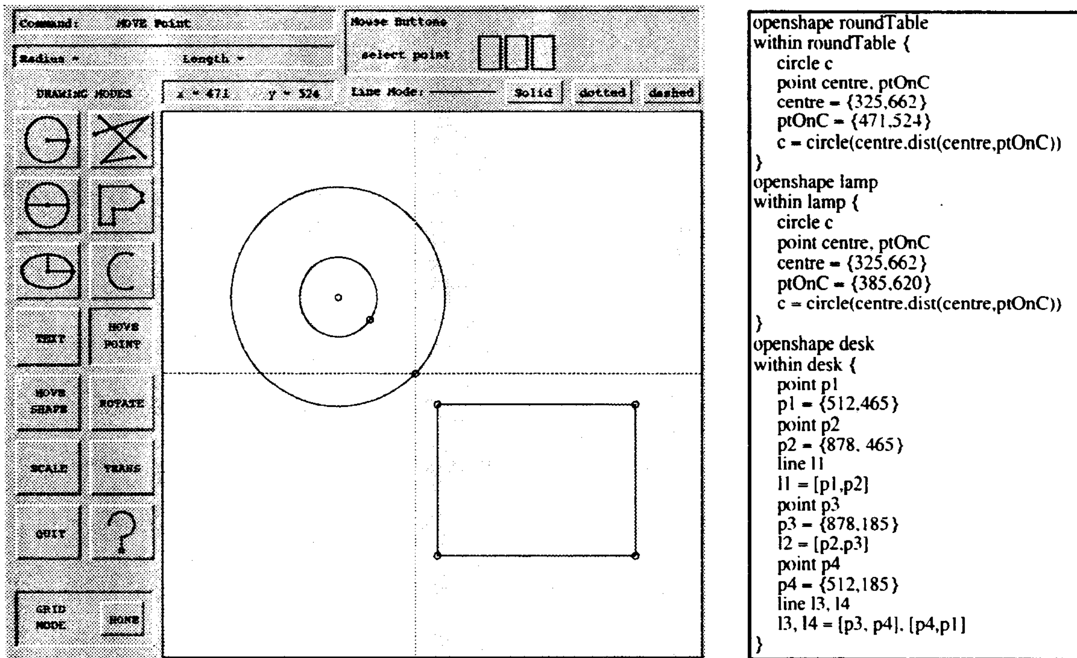
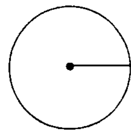


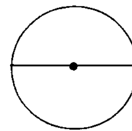
Figure 1

Left: The Graphical User-Interface of the Two-View Script Generator
Right: The DoNaLD Script Corresponding to the Drawing on the Left

As our main goal is to address a design task, we are led to consider the intended usage of the script in the larger context of design. The definitive script should give clues to the intended transformations of the object being designed. Conventional graphical interfaces achieve this in a limited way. Consider drawing a circle as an example. An application such as *xfig* displays icons such as:

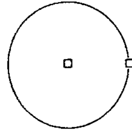


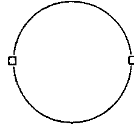
Icon 1



Icon 2

to indicate the modes of circle drawing. For Method 1 (with respect to Icon 1), a circle is drawn by giving the centre and a point on the circle; for Method 2 (with respect to Icon 2), a circle is drawn by providing the two extreme points on any diameter of the circle. Similarly, when modifying the specification of a circle, different sets of hot points will be shown. The specification is changed via moving of the hot points.





Hot points for Method 1 Hot points for Method 2

These icons and hot point settings reflect some common ways of modifying a design. However, the modification criteria are usually specific to a particular design. For example, whilst we may like to change the size of any individual gear in a gear box, it only makes sense to move a gear to a new position if all the gears are moved at the same time. Because of the application-specific nature of this requirement, it is not possible for us to pre-conceive all the possibilities for manipulating an object. As the gear example shows, some means to reference the attributes of other objects is required. This requirement is met in different ways by existing systems. ThingLab uses constraints to specify relationships between objects [BD86]; the DED graphics editor uses definitions to link up attributes between objects [Jeet87]. Constraints are usually resolved either by numerical methods, which may be very slow, or by selecting from a pre-defined set of methods. Each method is in effect a set of definitions in our terms. Our experience suggests that definition is a more primitive and effective means of representing objects. Furthermore, because a definitive script is developed through 'observation and experiment' it includes implicit information about which variables are likely to be redefined by external factors (i.e. which represent parameters that can be changed by the experimentor) and which are likely to be updated by the system itself (i.e. which represent associated responses to these changes).

Although a definition has a certain degree of persistence, and the variables more likely to be redefined are those at the leaves of the dependency tree, it is legitimate to make a radical change to the definition of any variable. For instance, if a lamp is sitting on a round table, we can denote this fact by writing the definition

lamp/centre = roundTable/centre

This has the effect that wherever the round table is moved, the lamp will move with it. The designer may later want to place the lamp on top of a desk. In that case, the variable *lamp/centre*, though not an independent variable, can be redefined to the centre of the desk (i.e. *(desk/p1 + desk/p3) div 2* in our example). The lamp location will then no longer be dependent on the table location.

As we have already indicated, a definitive script has advantages over a plain piece of drawing in that relationships between the components of the object are specified explicitly. A definitive script also gives clues as to which variables are more likely to be changed, but a script alone cannot restrict the scope for redefinition of any variable. This may be a desirable feature at an early or intermediate stage of design, where radical change to a design is frequent, but is not appropriate for *prescribing* the behaviour of the object under design. When we consider behaviour, we enter a new dimension for interaction with objects. This motivates the complementary part of our definitive framework – agent-oriented script management.

4. Agent-oriented Script Management

An agent in our framework can be a human agent or an autonomous agent with a circumscribed behaviour. An agent has certain privileges to perceive its environment and certain privileges to influence its environment. LSD is a notation we have developed for specifying agent privileges. In an LSD specification of an agent we specify the ***oracle*** variables which the agent can perceive, the ***handle*** variables which the agent can redefine, the ***state*** variables whose existence depends on the presence of the agent and the ***derivate*** variables which are defined in terms of other variables. There is also a ***protocol*** section to specify the actions the agent can perform when the state it perceives meets certain conditions.

The principles of LSD specification can be illustrated with reference to the implementation of our script generator prototype. Listing 1 shows a fragment of the LSD specification of some agents that are responsible for drawing and manipulating a circle using Method 1 described above. The agent *circle_handler* creates and manages the button for circle drawing. It keeps watch of the mouse activity by observing the oracle variable *circle_button_mouse*. (In our

underlying tool, when a Scout window is sensitive, such as the *circle_button* window, our system will redefine the variable *circle_button_mouse* to reflect the activity of the mouse in that window.) The agent *circle_handler* can instantiate agents *create_circle*, *create_manage_circle_hotpoint1* and *create_manage_circle_hotpoint2* if necessary. The agent *create_manage_circle_hotpoint1* not only creates a hot point (a sensitive region associated with an object) in the centre of the circle, but also manages it. Since the position of the hot point is indivisibly-linked to the centre of the circle, redefining the centre of the circle will move the hot point to the new position as well. When different parameters are given to the agents during instantiation, new sets of definitions and protocols are introduced. While the definitions and protocols introduced by the hotpoint managers are mainly to do with the interface, the definitions generated by the *create_circle* agent contribute to the initial design of the circle and the redefinitions caused by the hotpoint managers are modifications to the design. Of course, there is an uncircumscribed human agent (the user) in the system who can introduce new definitions or redefine existing definitions as he or she wishes.

```
agent circle_handler() {
    state
        string circle_name
        point circle_centre, point_on_circle
    handle
        string circle_name
        point circle_centre, point_on_circle
    oracle
        mstate circle_button_mouse                  // state of the mouse in the button region
    derivate
        scout
            // scout description of the button
            window circle_button = {
                type:       DONALD                  // show a DoNaLD picture on the button
                pict:       "circle_icon"           // the picture name
                box:        [{10, 100}, {60, 150}]  // geometry of the button
                sensitive:  ON                      // a sensitive button
            }
        donald
            // donald description of the icon
            circle_icon = ...
    protocol
        pressed(circle_button_mouse) ->
            circle_name = user_input(string);
            circle_centre = user_input(point);
            point_on_circle = user_input(point);
            create_circle(circle_name)(circle_centre, circle_end);
            create_manage_circle_hotpoint1(circle_name);
            create_manage_circle_hotpoint2(circle_name)
}
```

```
agent create_circle(?name)(?centre, ?ptOnC) {
    derivate
        donald
            openshape ?name
            within ?name {
                circle c
                point centre, ptOnC
                centre = ?centre
                ptOnC = ?ptOnC
                c = circle(centre, dist(centre, ptOnC))
            }
}

agent create_manage_circle_hotpoint1(?name) {
    handle
        point ?name/centre                           // position of the centre of the circle
    oracle
        mstate ?name_hp1_mouse                       // state of mouse action within the hotpoint
        point ?name_hp1_mouse_position               // relative to the centre of hotpoint
    derivate
        scout
            window ?name_hp1 {
                type:       DONALD
                pict:       "NULL"                   // empty picture
                box:        [?name/centre - {2, 2}, ?name/centre + {2, 2}]
                border:     1                        // a plain box only
                sensitive:  ON
            }
    protocol
        button1_released(?name_hp1_mouse) ->         // move the circle
            ?name/centre = | ?name/centre + position(?name_hp1_mouse) |
}

agent create_manage_circle_hotpoint2(?name) {
    handle
        point ?name/centre                           // position of the centre of the circle
        point ?name/ptOnC                            // position of the reference point on the circle
    oracle
        mstate ?name_hp2_mouse                       // state of mouse action within the hotpoint
        point ?name_hp2_mouse_position               // relative to the centre of hotpoint
    derivate
        scout
            window ?name_hp2 {
                type:       DONALD
                pict:       "NULL"                   // empty picture
                box:        [?name/ptOnC - {2, 2}, ?name/ptOnC + {2, 2}]
                border:     1                        // a plain box only
                sensitive:  ON
            }
    protocol
        button1_released(?name_hp2_mouse) ->         // move the circle
            ?name/centre = | ?name/centre + position(?name_hp2_mouse) |
        button2_released(?name_hp2_mouse) ->         // move the point on the circle
            ?name/ptOnC = | ?name/ptOnC + position(?name_hp2_mouse) |
}
```

Listing 1: A Fragment of the LSD Specification for the Two-View Script Editor

Notice that an LSD agent specification places restrictions upon how changes of state can occur, but does not itself determine the behaviour of the agents. It is only when the pattern of the agent behaviour is preconceived that the agent specifications can be interpreted as autonomous agents. Animation is then possible through systematic execution of their privileges. By supplying assumptions required for simulation, such as the relative speed of action execution, the agents' choice of action when several actions are legitimate, and the accuracy of agents' perception, the LSD specification can be transformed into an executable program. We have developed many examples to illustrate these principles of animation. These include: a train departure and arrival protocol simulation, a vehicle cruise control simulation and a sail-boat simulation [NBY94, BBY92, BY92].

We have developed two computational models for the implementation of LSD specifications – ADM and EDEN. Central to ADM is a Definition Store, an Action Store and an Entity Store. An entity in ADM resembles a transformed LSD agent. An entity is a group of definitions and actions, and an action is a guarded sequence of primitive commands, where a primitive command is either a redefinition, or the instantiation or deletion of an entity instance. All active actions are kept in the Action Store. In each execution cycle, the commands with true guards are executed in parallel, causing changes to the Definition Store or the Action Store or both.

The alternative computational model, EDEN, is more procedural in style. Perhaps the most important feature of EDEN in relation to implementation of LSD is the EDEN action. An EDEN action is a C-like procedure (with definitive variables) that is triggered when the values of one or more of the specified triggering variables are updated. This characteristic of EDEN actions makes EDEN ideal for simulating an LSD specification in which all agents other than the human user of the system are passive (event-driven). For instance, EDEN is used for implementing the vehicle cruise control simulation and the sail-boat simulation. In contrast, the ADM model is used for implementing the train departure and arrival protocol simulation, since this involves some active agents like passengers, a station master and a train driver.

The two-view editing system illustrated above also involves several passive agents. In the graphical editor, there is an agent behind each button that will perform the intended action as

soon as it perceives an appropriate mouse action. As an event-driven system, the two-view editor is implemented in EDEN.

5. Current State of the Project and Concluding Remarks

The two-view script generator is still under development. As yet, relatively few operations are defined in the editor. An important feature, currently under development, is a query button to display the set of definitions representing a selected object. One of the difficulties in using scripts in design is that of identifying the references to an object on the display. The query function will assist the user in finding out the references to an object and in specifying potential transformations. This makes it easier for the user to introduce relationships between objects.

In a previous paper, we described a role for agent-oriented definitive scripts in concurrent engineering design. In this paper we have described a means to the computer-aided generation of such scripts. Although the script that is generated in this manner only corresponds to a small fragment of the whole design, and is restricted to simple line drawing, it represents a significant contribution to the overall design process. In addition, the two-view script generator is itself written using a definitive script. This shows the practicality of definitive programming.

Acknowledgement

We are indebted to Mark C K Leung for prototyping the two-view script generator. We are also grateful to the SERC for financial support under grant GR/J13458.

References

[ABCY94] Adzhiev, V., Beynon, W. M., Carwright, A. J., Yung, Y. P. *A Computational Model for Multi-Agent Interaction in Concurrent Engineering*, Proc. CEEDA 94, to appear

[BBY92] Beynon, W. M., Bridge, I., Yung, Y. P. *Agent-Oriented Modelling for a Vehicle Cruise Control System*, Proc. ASME Conf. ESDA '92, Istanbul 1992, 159-165

[BD86] Borning, A, Duisberg, R. *Constraint-Based Tools for Building User Interface*, ACM Transactions on Graphics, Vol. 5, No. 4, Oct 1986, 345-374

[Beynon89] Beynon, M. *Evaluating Definitive Principles for Interactive Graphics, New Advances in Computer Graphics*, Springer-Verlag 1989, 291-303

[BR92] Beynon, M., Russ, S. *The Interpretation of States: A New Foundation for Computation*, Computer Science Research Report 207, University of Warwick, 1992

[Brooks91] Brooks, K. P. *Lilac: A Two-View Document Editor*, Computer, June 1991, 7-19

[BY90] Beynon, W. M., Yung, Y. P. *Definitive Interfaces as a Visualisation Mechanism*, Proc GI '90, Canadian Inf Proc Soc, 1990, 285-292

[BY92] Beynon, W. M., Yung, Y. P. *Agent-Oriented Modelling for Discrete-Event Systems*, Proc IEE Coll. "Discrete-Event Dynamic Systems", Digest #1992/138, June 1992

[CW89] Chmilar, M., Wyvill, B. *A Software Architecture for Integrated Modelling and Animation*, New Advances in Computer Graphics, Proc. of CGI'89, 257-276

[EDC93] *A Short Guide to the Engineering Design Centre and Its Work*, Lancaster University Engineering Design Centre, 1993

[Jeet87] Jeet, E. J. *A Relationship-Based Interactive Graphical Diagram Editor*, PhD thesis, Department of Computer Science, University of Kent, Nov 1987

[JNZM93] Johnson, J. A., Nardi, B. A., Zarmer, C. L., Miller, J. R. ACE*: Building Interactive Graphical Applications*, Communications of the ACM, vol. 36, No. 4, April 1993, 41-55

[NBY94] Ness, P,. Beynon, W. M., Yung, Y. P. *Agent-oriented Modelling for a Sail Boat Simulation*, Proc. ASME ESDA '94, London, to appear

[Wyvill75] Wyvill, B. *An Interactive Graphics Language*, PhD Thesis, University of Bradford, 1975