Computer-Assisted Jigsaw Construction: a Case-Study in Empirical Modelling

W.M. Beynon, C.J. Sidebotham, Y.P. Yung
Department of Computer Science
University of Warwick
Coventry CV4 7AL
UK

Abstract

We connect the problem of finding a suitable programming paradigm for graphics with its role in the visualisation of *conceptual representations* – state-based models that can capture an experiential view of knowledge that arguably cannot be expressed in a logical or declarative framework. Conceptual representations are identified by selecting systems of observables, and analysing the functional dependencies through which they are indivisibly linked in change. Such representations have a particularly significant role to play in exploratory design – an activity that has a high profile in modern computing applications such as modelling and simulation for virtual reality, reactive systems and concurrent engineering.

Because it links programming and modelling, an object-oriented paradigm offers some support for conceptual representation, but the emphasis it places upon object integrity, circumscription and encapsulation is in general too restrictive. We introduce an alternative modelling paradigm – *empirical modelling* – that is well-suited for conceptual representation in exploratory design, and outline its application to modelling the requirement for a computer-assisted jigsaw assembly environment. In this way, we illustrate the use of a software environment, incorporating three definitive (definition-based) notations for specifying graphical displays, that we have developed to construct and visualise conceptual representations.

1. Introduction

Computer graphics presents a particular challenge to the theory of computer science. The problem of finding an appropriate programming paradigm for graphics is symptomatic of this. In this paper, we connect this problem with the use of computer graphics to represent state in a direct metaphorical fashion, in a sense to be discussed below. Declarative programming paradigms rely too heavily upon linguistic frameworks and on modelling state through circumscription to be well-adapted for this. Object-oriented programming is to an extent better suited for this purpose, but is most effective only when a high degree of conviction about the potential transformations of state to be represented has been acquired. The solution examined in this paper is based on a new empirical modelling approach.

2. Empirical Modelling

The purpose of this paper is to introduce and illustrate the application of a new method of modelling that is motivated by the same considerations that first motivated object-oriented programming [4] – in brief, the concept that programming is a form of modelling. Our modelling method – empirical modelling – is concerned with relating two views of knowledge: experiential and theoretical. By way of clarification, an experiential view of knowledge is illustrated when we first experiment with an object such as Rubik's cube, which exhibits many different states and can be transformed in many different ways. A theoretical view of same domain of knowledge of the

behaviour of the cube is expressed in established transformation patterns, such as those that we memorise in solving the cube. This paper is centrally concerned with techniques for the representation of experiential knowledge – an area in which visualisation through computer graphics has a crucial role to play.

The essential focus of empirical modelling reflects the dictionary definition of empirical: "that which is based upon observation and experiment". Empirical modelling addresses the process of transforming from experiential to theoretical perspectives. Its philosophical stance is a form of empiricism that is concerned with tracing knowledge to its roots in experience. The extent to which an experiential view of knowledge can be transformed to a theoretical view depends both upon the domain and the observer. Not all experience can be captured by theory. Consider, for example, how the directions to a landmark in a city might be expressed in terms of observations that are particular and personal ("you'll see a fishmonger with a big red scarf on the corner") that will not appear on any map.

Experiment is the key concept in empirical modelling. It provides the mechanism by which an experiential view of knowledge is converted into theoretical view. The nature of this conversion process is subtle: it is associated with reinterpretation on the part of the experimenter rather than newly acquired experience. This is most simply expressed by the paradoxical nature of experiment:

- A Good Experiment is one in which the outcome is uncertain
- A Good Experiment is one in which the outcome is entirely predictable.

A pattern of experience (as defined by a scenario of action and response) can be viewed as a theory when the experimenter has conviction that it will be reliably observed.

Computer Science is well-versed in the representation of theoretical knowledge. Logic, as mediated through formal languages with a well-defined operational semantics, supplies the conventional framework for this representation.

The representation of experiential knowledge, such as an experimenter acquires empirically, is (by contrast) inadequately addressed by theoretic computer science. In our view, the appropriate paradigm for this kind of knowledge representation is that employed by the craftsman or the engineer, who traditionally constructs a physical model that makes experience accessible through imitation. Unlike a document, such a model is not linguistic in nature but metaphorically represents one amongst many external states that can be explored through observation and experiment.

The basic principle of empirical modelling is the construction of *conceptual representations* of real-world phenomena. Such representations are defined by the selecting a set of observables and – by analysing the indivisible relationships between these observables through experiment – identifying the functional dependencies amongst them. In the classical engineering context, conceptual representations are typically embodied in a physical model (e.g. a scale model of a mechanical device), but can also be more abstract in nature (e.g. a circuit diagram metaphorically represents a configuration of observables). The advent of computers and multimedia has enormously enhanced the potential for the abstract construction (as in the spreadsheet) and physical realisation (as in a virtual reality environment) of conceptual representations. The representation techniques we have developed for empirical modelling exploit visualisations of generalised spreadsheets based on scripts of definitions (definitive scripts), as illustrated in §4.

When realised in a physical model, conceptual representations exhibit features that correspond directly to external observables in the experimental context to which they refer, and can be observed and realised in many different configurations. Their exploitation relies upon establishing a close correspondence between the outcome of experiments on the model and in the associated real-world environment (thought experiments may be involved in either or both contexts). Empirical modelling is the process by which parallel experiments in these two contexts are correlated. It leads to the refinement of conceptual representations, either through a more precise articulation of dependencies, or through the introduction of additional observables.

Its emphasis upon imitating indivisible relationships between observables distinguishes empirical modelling from object-oriented modelling. (In object-oriented modelling, observables are grouped together into local state-transition models, without regard for whether changes in observables propagate across object boundaries.) Indivisibility can play a powerful role in establishing the connection between a state-based model and its real-world referent – a feature that associates form and content in a way that is beyond the scope of logical representations. Our concept of indivisibility does not require that the changes that occur within a model should be instantaneously synchronised (cf the way in which spreadsheet updates propagate), but that this propagation cannot be suppressed and that there is no ambiguity about which states of the model are to be interpreted.

The significance of graphics in modern computing stems from three factors:

- The increasing prominence of applications in which the computer is used to embody conceptual representations of external state,
- the importance of sight as our primary sensory channel,
- the pre-eminent role of the screen interface in making the internal state of a computer perceptible.

In effect, modern applications put a premium upon being able to construct graphical displays in which the functional dependencies between geometric features directly imitate dependencies between external observables. Our empirical modelling software tools are concerned with constructing geometric models of just this nature.

Computer models based solely on graphical conceptual representations are typically only components of larger systems. Their role may for instance resemble that of a scientific instrument, such as a speedometer, in a reactive system. Such models belong to a broader class of direct representations (including virtual reality systems) and can incorporate embedded linguistic components (such as messages) whose presence and content may also be determined by functional dependency. Roadside hazard warning notices that are triggered when a high vehicle approaches a low bridge provide one example of this kind. Visualisation of conceptual representations is not only significant where human interaction is explicitly involved – some form of visualisation is also arguably essential in the design of any complex system[7].

The development of models based on conceptual representations differs from that of theoretical models. It is necessarily constructive by nature, so that the evolution of the model accompanies understanding and insight. In this process, we can draw upon pre-established conventions (cf map construction), but cannot rely upon them exclusively.

The process involves more than developing conventions in general – we must conceive a physical device that reliably displays an appropriate state-changing behaviour.

3. The Case-Study

The principles of empirical modelling will be illustrated with reference to a case-study: viz. requirements modelling for a computer-assisted jigsaw assembly environment.

There are several reasons for this choice of case-study:

- requirements modelling tasks are well-suited for empirical modelling techniques, since they involve the representation of tentative concepts that require frequent extension and modification. Annotated graphical models can play a most significant role in the representation process, allowing us to construct explicit state-based environments for interaction in which to imitate experiences that (in view of their uncircumscribed nature) defy explicit description in a logical or theoretical framework. In such applications of graphics, object-oriented programming is a cumbersome tool that forces the designer to circumscribe the behaviour of objects prematurely and to establish artificial barriers around objects through encapsulation.

- the process of jigsaw assembly is itself an open-ended task with many characteristics in common with a design activity. It combines high-level and low-level aspects, in which a human agent operates in both conceptual and technical frames of reference. It has exploratory and experimental aspects, and involves milestones and commitments. It involves perpetually shifting contexts, as the "design artefact" (the partially completed jigsaw) and the environment (represented by the assembler's view of the jigsaw) evolve. It is also a process in which human cognition and interaction plays an essential role. (For instance, the most common strategy for jigsaw puzzle solution: "locate and assemble all the significant and meaningful pieces first and leave the background and vague pieces to last" would in general involve human intelligence.)

- the characteristics of a jigsaw assembly environment cannot be determined entirely without reference to the specific jigsaw to be solved (nor indeed the specific human assembler). This effectively means that a suitable jigsaw assembly environment must allow the assembler to customise the generic environment as initially supplied – this customisation process is itself similar in character to the requirements modelling exercise as a whole. For this reason, the modelling paradigm we introduce may also be appropriate for the implementation of a jigsaw assembly environment.

The fact that jigsaw puzzles are essentially visual of course lends graphical interest to the case-study, but the issues it raises are of broad relevance to all applications in which design and conceptual representations are involved. The complexity of the human cognitive processes surrounding the evolving model demand a framework within which conceptual representations, and graphical representations in particular, can be freely integrated into an environment that in general also includes textual and direct manipulation ingredients. Similar considerations and principles would apply to the requirements specification for a musical jigsaw for instance, in which the role of the jigsaw pieces was played by musical fragments, and the objective was to assemble these fragments to realise a given aural effect.

Preliminary work on the jigsaw assembly environment case-study has been carried out by Cheryl Sidebotham as her final-year undergraduate project since October 1994. This has led to the development of a prototype "vanilla" environment in which a simplified computer-based jigsaw puzzle can be automatically constructed from a coloured 2D line drawing by automatic subdivision into square pieces, which can then be manipulated by the human assembler through a suitable interface to assemble the jigsaw (Figure 1). The assembly process is monitored by recording the combinatorial relationships between pieces that make up the partially completed jigsaw.

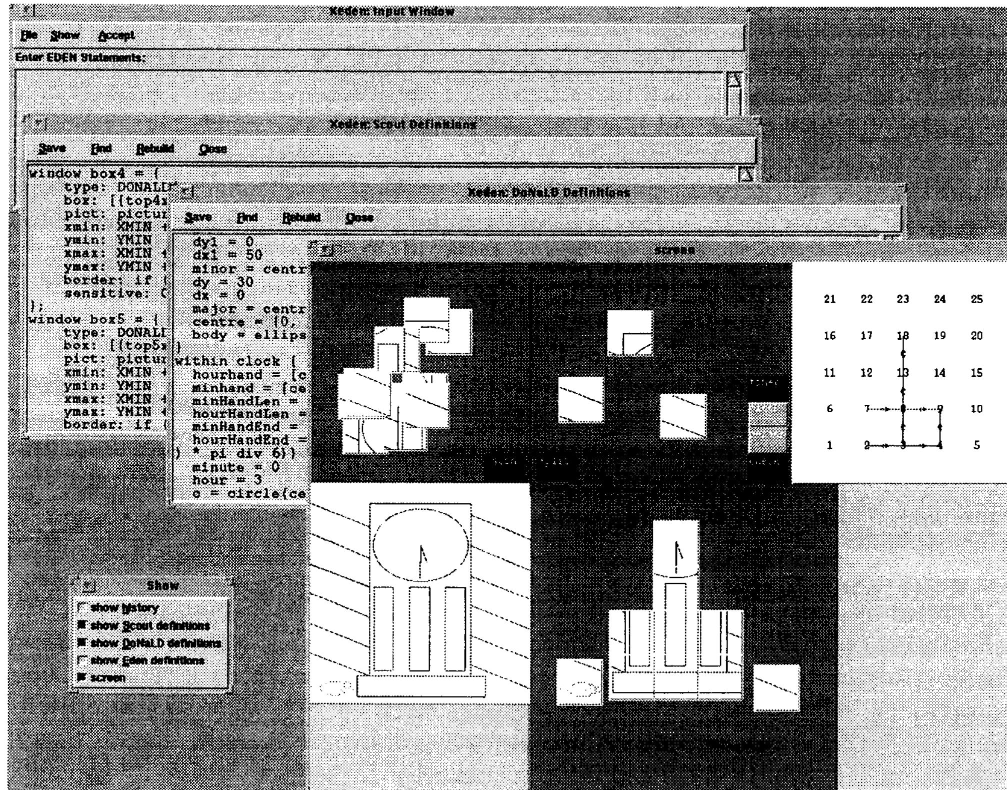

Figure 1

The modelling process used to construct the environment in Figure 1 is very open-ended, and Figure 1 is just one of several possible variant designs that have been derived in an exploratory fashion. Variants that can readily be derived by relatively minor modification of the vanilla environment include jigsaw assembly environments in which:

- pieces can be displayed in many different orientations, including upside-down
- the jigsaw picture is dynamic, so that the pictorial content of pieces changes whilst assembly is in progress.
- the pieces are rectangular in shape.

The full potential for a computer-based jigsaw assembly environment is harder to conceive. In one of several different directions of development, we might envisage being able to transport a commercial jigsaw (with several hundred pieces) into a computer environment, so that it can be manually assembled with computer assistance in classifying and selecting pieces according to criteria introduced dynamically during assembly. In principle, it would then be possible for the assembler to mark subsets of the pieces according to their characteristic properties (e.g. pieces that are uniformly coloured, or form part of a pictorial feature etc), and to invoke automatic retrieval

according to the classification established in this way. It should be noted that the aim of developing a jigsaw assembly environment is not to automate jigsaw construction to such a degree that it becomes a routine process for the assembler (cf [6,9,10,13]), but to automate the tedious and routine aspects of the task.

4. The Tools

The model we have constructed is based on a set of modelling/programming tools designed and implemented locally in the University of Warwick, primarily by Y.P. Yung. These tools are built upon a common principle of representing state using a set of definitions. A *definitive state* (a set of definitions) differs from states in ordinary procedural or object languages in that relationships between variables are implicit in the *definitions* of the variables. In this respect, definitive variables resemble spreadsheet cells. A definition such as "Y = 2 * X + C" is taken to mean that as long as this definition holds, Y will be maintained to be 2X+C. That is, whenever X or C changes value, so will the value of Y. This declarative nature of the definitive variables entails less commitment than is presumed when defining variables in many functional or constraint-based languages, however. A definitive variable can be redefined both during model execution and in model revision. Persistent relationships subject to revision of this kind are highly suitable for modelling empirical results such as the intermediate states of the jigsaw puzzle assembly. Definition-like principles have been applied to computer graphics before (cf [5]), but we apply them to a more general modelling framework. The way in which we apply the principles in modelling the process of jigsaw assembly will be dealt with in the next section. This section describes the tools that facilitate the modelling process.

Our set of tools comprises an interpreter for a general purpose definitive language EDEN and several translators which transform definitions in higher-level application domains to definitions and other supporting statements in EDEN. In the jigsaw example, we have incorporated three definitive (definition-based) notations: Scout, DoNaLD and ARCA. Each of these addresses a certain aspect of the jigsaw assembly problem. DoNaLD is a definitive notation for line drawing. It is used to describe the jigsaw picture. ARCA is a definitive notation for specifying and displaying combinatorial graphs. It describes the symbolic interconnection between the jigsaw pieces in the partial solution. Scout is a definitive notation for describing screen layout. The window layout and the buttons are specified in this notation.

The simplest way to understand a definitive notation is to refer to examples. For this purpose, the 2D line drawing notation DoNaLD may be the best starting point. Listing 1 is a DoNaLD description of a simple clock face consisting of a circle representing the perimeter of the clock and two lines representing the two hands set at the 3 o'clock position. It may be helpful to point out at the outset that the *within* clause is more like a scoping convention – similar to that in the Unix file system – than an encapsulation convention in OOP. That is:

c = circle(centre, radius)

in this example has the same meaning as defining:

clock/c = circle(clock/centre, clock/radius)

```
openshape clock
within clock {
 line hourhand, minhand
 point centre, minHandEnd, hourHandEnd
 real radius, minHandLen, hourHandLen
 int minute, hour
 circle c

 hourhand = [centre, hourHandEnd]
 minhand = [centre, minHandEnd]
 minHandLen = radius * 0.9
 hourHandLen = radius div 2
 minHandEnd = centre + {minHandLen @ (pi div 2 - minute * 2 * pi div 60)}
 hourHandEnd = centre + {hourHandLen @ (pi div 2 - (hour + minute div 60.0) *
       2 * pi div 12)}
 c = circle(centre, radius)
 minute = 0
 hour = 3
 radius = 120.0
 centre = {460, 720}
}
```

Listing 1

For that matter, *clock/c* can equally well be defined in terms of other points associated with another openshape variable. The essential function of a DoNaLD script is therefore not to circumscribe an object boundary but to declare the types of variables and their interrelationships. Variables can be defined in any order and can also be redefined later. Redefinitions are possible for both variables defined by explicit values and variables defined in terms of other variables. In this script for instance, one can simply redefine *radius* in order to resize the entire clock. The redefinition of *radius* will directly or indirectly affect the values of almost all other variables. Changes to the variables *hour* and *minute* have a similar but less profound effect. These will change the position of the hands only. (Of course, we usually introduce a clocking agent to update the variables regularly. The role of agents in our models will be elaborated in the next section.) More complicated redefinitions are also possible. One can redefine *hourHandEnd* to:

```
hourHandEnd = centre + {hourHandLen @ (pi div 2 - hour * 2 * pi div 24)}
```

so that the clock is now a 24-hour clock. If so desired, one can introduce a new variable to make the selection of clock type an option. For example:

```
bool twelveHours
hourHandEnd = centre + {hourHandLen @ (pi div 2 - hour * 2 * pi div (if
       twelveHours then 12 else 24))}
```

All these changes can be done on-the-fly. On redefining a variable, our underlying definition manager will automatically update the values of the variables dependent on the changed variable and hence re-visualise them on the screen. The net result is that, at any time, the graphical display will always reflect the state of the set of definitions in store.

The concept of indivisible propagation of state change allows us to look at definitions in a different light. We can now consider a set of definitions as a representation of states and redefinitions as state transitions. All our definitive notations are interpreted in a manner similar to that of DoNaLD, but use different underlying algebras. Instead of *point*s, *line*s, *arc*s and *circle*s etc., ARCA allows us to define variables of types *vertex* and *colour* (coloured edges) using special combinatorial operators, while Scout allows us to specify regions and the contents of the regions. An example of a Scout window is given in Listing 2 – it defines a Scout window that represents the 4th of the 25 jigsaw pieces.

```
point dim = {50, 50}
integer width = (XMAX - XMIN) / 5
integer height = (YMAX - YMIN) / 5
window piece4 = {
    type:       DONALD
    box:        [ {top4x, top4y}, {top4x, top4y}+dim ]
    pict:       pictureStr
    xmin:       XMIN + 3*width
    ymin:       YMIN
    xmax:       XMIN + 4*width
    ymax:       YMIN + height
    border:     if (selection1==4 || selection2==4) then 2 else 1 endif
    sensitive:  ON
};
```

Listing 2

A Scout window has different attributes according to its *type* – the kind of information to be displayed. In this piece4 window, a DoNaLD drawing is to be displayed. The size and location of the window is defined by the *box* attribute. The *pict* attribute determines which DoNaLD drawing (a collection of DoNaLD variables) is to be displayed. In this case, a string variable *pictureStr* holds the name of a DoNaLD drawing. By changing this variable, we can switch to another jigsaw puzzle. This function can be performed using the buttons *person* and *clock* which essentially do nothing more than redefine the *pictureStr* variable. The *xmin*, *ymin*, *xmax* and *ymax* attributes define how the picture should map onto the region defined by the *box* attribute. It is then obvious that we are defining all 25 pieces of the jigsaw as different visualisations of the same drawing. For this reason, we are able to cope with dynamic changing of the jigsaw picture without having shown foresight about the possibility of its happening.

In Listing 2, *border* (thickness) is defined by a conditional expression. The jigsaw assembly system allows the assembler to select up to two pieces of jigsaw at the same time. These selected pieces will be highlighted by the thickening of the window borders. The *sensitive* attribute determines whether mouse or keyboard activity within the window should generate a redefinition. The variable to be redefined is related to the Scout window name. Should a mouse button be pressed inside *piece4* for instance, the variable *piece4_mouse* will be redefined to a compound value containing mouse information such as which button is pressed and the location of mouse click. There is an agent, implemented in the EDEN language, that monitors changes to such variable and reacts appropriately (e.g. by selecting piece4 or relocating the piece by redefining *top4x* and *top4y*).

The reason why scripts of different definitive notations can be integrated in our framework is that all these definitions of different notations are translated to the same general purpose definitive language EDEN, and so are maintained by the same definition manager. EDEN definitions can therefore be written to serve as linkages (sometimes called *bridging definitions*) between variables of different definitive notations. EDEN has arithmetic types, strings and heterogeneous lists, which can simulate very complex data types. It also allows us to define functions and procedures in C-like fashion. We can also specify some procedures to be automatically executed in response to changes to some variables. This type of triggered procedure (or *action*) provides basic support for agent observation and response and is used to synchronise the visual states with the definitive states.

Figure 1 shows a sample screen display of our jigsaw assembly environment in action. This environment not only allows the assembler to interact via the jigsaw assembly interface (bottom-right window), it also allows the assembler to enter any statements that are acceptable by EDEN, Scout or DoNaLD through the input window (in the top-left corner). (Our prototype does not support ARCA directly at present.) Our environment also allows the user to look at the current definitive state from different angles. For instance, the windows behind the jigsaw display reflect the Scout and DoNaLD definitions that have been processed by the associated definitive translators. Note that these definitions do not necessarily reflect the current state of the definitive machine. For instance, readers with good eyesight can read that the time recorded in the DoNaLD translator is 3 o'clock while the actual time on the display is 5:30. This discrepancy arises because after DoNaLD has translated the definitions of *clock/minute* and *clock/hour*, the translated EDEN images of these variables are redefined so that they are now linked to another EDEN variable denoting the current time. (This device facilitates the motion of the clock.) There is potential for confusion here as to whether we need to work with the translated script in order to know the current status of definitions. A colour convention in the EDEN definition view is used to resolve this confusion. In that view, definitions introduced via the DoNaLD translator appear in a different colour from those directly defined in EDEN. These definitions can in fact be eliminated from the view entirely, leaving only the EDEN redefinitions behind.

In summary, we have described a development environment for models based on definitive representation of states. This environment provides facilities for user interaction, for introducing unforeseen requirements, and for examining and storing the current state of development.

5. The Development Process

The development of the requirements model is based upon a close analysis of the jigsaw assembly process. The analysis is agent-oriented, in that the sensory and cognitive elements involved in jigsaw assembly are expressed in terms of viewpoints and privileges from which the entire activity can be synthesised. In this context, our use of the term *agent* is similar in spirit to that introduced in Minsky's The Society of Mind [12]: for instance, the role of the assembler is interpreted with reference to primitive perceptions and actions that contribute to successful completion of the assembly task. In its emphasis on viewpoint, our approach also resembles subject-oriented programming, as introduced in [8].

At its present stage of development, our jigsaw assembly environment addresses only the low-level aspects of jigsaw specification and assembly. These concern the mechanics of setting up the jigsaw, of moving and joining pieces and manipulating

fragments of the partially completed jigsaw. Analysis of the low-level aspects leads us to consider such questions as:

- how are pieces distinguished?
- how are pieces matched?
- when are pieces regarded as joined?
- how is the current completion status of the jigsaw assessed?

Close analysis of the physical capabilities of the assembler is directly relevant to conceiving the jigsaw assembly environment. Whether pieces are presented in a heap depends upon whether the assembler can distinguish one piece from another and move them apart. In matching pieces it is appropriate to place them side by side, or to place them on the jigsaw picture. In principle, our modelling methods offer great flexibility, allowing us to prototype jigsaw-related activities that invest the environment and the assembler with unusual characteristics. For instance, it is trivial to set up the control environment for our existing model in such a way that the clock picture is frozen at a time that can be freely selected by the user at any stage, yet the pictorial elements on the pieces still change dynamically with time. Alternatively, the jigsaw can be set up in such a way that the pieces are presented face-down in a grid arrangement, as in a game of pelmanism, so that their selection leads to their temporary inversion. In this scenario, an upturned piece would be removed only if it could be correctly placed the jigsaw .

In practice, the present limitations of our tools make some scenarios of interaction more convenient to implement than others. Scout windows can only be oriented to be parallel to the axes, for instance. When pieces are re-oriented, it is difficult to keep track of their attributes. The close connection between low-level human processing and the functionality of the requirements model is illustrated by way that limitations of the tools suggest constraints on the environment and profiles for the assembler. For instance, magnetised pieces might be restricted in their orientation, and coloured pieces would be monochrome in certain orientations if viewed through polarised lenses.

In our prototype environment, the higher level knowledge concerning whether pieces are joined, and which jigsaw fragments have been constructed, is modelled in EDEN and visualised in the ARCA diagram. Two pieces that are placed side by side are joined by an explicit action on the part of the assembler, associated with selection of the **join** button. The combinatorial relationships recorded in the ARCA diagram make it possible to treat each completed fragment of the jigsaw as a generalised piece that can be moved by selection of any of its constituent pieces. When the assembler joins a piece to a jigsaw fragment in such a way that it simultaneously abuts more than one piece, more than one adjacency relation must be introduced to the ARCA diagram. This can be most conveniently handled automatically by adding agents in EDEN to perform this action when required.

The ARCA diagram records information about the state of completion of the jigsaw. It can also be used to check whether the jigsaw is correctly completed, and to monitor the correctness of the assembler's actions. Different strategies for using this information might be used to change the character of the assembly task. For instance, the assembler might be barred from making false construction steps. It is a feature of our modelling environment that the high-level control structures that govern the behaviour of the model can be relatively easily reconfigured, even on-the-fly, through introducing appropriate agents with relatively simple protocols for guarded redefinition. The simplicity of such agents depends greatly upon the fact that the effect of redefinition upon system state is context-dependent, as mediated by the definitions extant in the

context for execution. This eliminates the need for wholesale reprogramming of agents to introduce new side-effects of existing actions.

The role of the ARCA diagram in our model is subtle. In the context of our vanilla environment, where the pieces are square, the interconnection is very simple and easy to realise as a combinatorial graph. The diagram itself should not of course in general be visible to the assembler – its main value has proved to be in supplying useful visualisation for the modeller, making it possible to detect problems arising in the model development. The link between the geometry of the jigsaw and the structure of the ARCA diagram becomes more obscure when rectangular pieces are introduced, and also raises issues concerning dependency that are yet to be adequately addressed by our tools. Ideally, one would like the insertion of a piece and the insertion of an edge to be represented – as are other indivisibly associated actions – as a form of redefinition in the model, but in fact an action has to be invoked to establish this link.

The greatest challenge to our modelling approach is presented by the cognitive aspects of jigsaw assembly that impinge when we consider strategies the assembler might use to identify and classify pieces. In this connection, the idea that a jigsaw piece has more content than can possibly be expressed in a circumscribed model is clear. For instance, the assembly environment, acuity of physical skills, memory, familiarity with the jigsaw, knowledge of the semantics of the picture etc all influence whether an assembler can perceive the characteristics and relationships relating to jigsaw pieces. The extent to which a jigsaw assembly environment can allow the assembler to invoke automatic assistance in respect of such issues is difficult to assess. It is evident however that any satisfactory solution must exploit human-computer cooperation, e.g. allowing the machine to apply an approximate selection criterion, whilst giving the assembler scope to arbitrate in borderline cases.

Even relatively simple forms of classification involve functional dependencies that defy expression in terms of our existing notations. For instance, "identifying all the pieces that have red on them" is clearly a functional dependency, but requires operators outside our existing underlying algebras for precise specification. An appropriate way to specify such pieces would be to determine the list of identifiers of red geometric elements in DoNaLD, then to identify which pieces of the jigsaw these entities intersect. We at present have no syntactic framework for representing such dependencies. A simpler and less precise characterisation of red pieces that has incidental interest is defined by the set of Scout jigsaw piece windows (cf Listing 2) that contain a red pixel. Such a characterisation depends upon the resolution of the screen display, and corresponds metaphorically to an assembler working at a distance.

6. Object-Oriented Modelling and Conceptual Representation

The object-oriented programming paradigm, as originally expounded by Birtwistle et al (Simula 67) was directly concerned with issues of conceptual representation. The philosophy behind "programming as modelling" places the emphasis upon computer programming as a form of system description. In Simula, objects are conceived as corresponding to situated physical entities that interact with their environment, and are constructed through analysis of the observables to which they react and through which they respond. Object-oriented modelling has certain useful qualities for conceptual representation. Encapsulation and methods can be interpreted as means to specify synchronised change to observables (in as much as updates to procedural variables are grouped within a method and presented externally as one conceptual state-transition). In so far as Listing 1 identifies a clock with a conceptual representation specified in

terms of a fixed set of observables and transformations, a similar function of specifying a clock as a geometric object can be served by an object-oriented model. However, object-oriented principles would not assist the designer in making opportunistic changes to the clock specification such as can be effected on-the-fly through redefinition, nor allow changes to the clock to be conveniently synchronised with other state-changing mechanisms subsequently introduced into the model.

The limitations of object-oriented programming (as presently conceived) in respect of conceptual representation are largely consequences of the way in which the fundamental abstraction (the "object") has been developed without sufficient reference to the original context and motivation for its introduction. It is natural to associate observables into local pieces of state whose presence in the environment reflects the existence of particular entities, but too great an emphasis upon object specification and integrity is obstructive. Considerations include the following:

- experience is acquired (in the first instance) through interacting with instances of objects, not classes. Modelling the relationships between observables should be primarily guided by what is observed, not based upon the premature commitment to generic patterns.
- the set of observables associated with an entity cannot be circumscribed. Circumscription of the set of observables associated with an entity only makes sense in relation to particular functions it serves, subject to empirically acquired conviction that all relevant attributes have been ascertained. (For instance, the significant attributes of a jigsaw piece can surely never be completely circumscribed, since they depend on altogether subjective perceptions and personal knowledge of the assembler.)
- atomic changes to observables in a system are not in general localised to objects.

The concept of object-oriented modelling is sufficiently broad, and ill-defined, that none of the issues raised above is incompatible with an object-based framework. It is the development of object-oriented programming as a paradigm for modular software construction, its applications to distributed software development and its emphasis on generic objects as a route to reuse that has distracted attention from the agenda suggested by these issues.

To restore an object concept that is better suited to conceptual representation (cf [11]) requires a shift in perspective, whereby we acknowledge that in general

- neither the observables nor the possible transformations associated with real-world objects can be conveniently circumscribed
- the classification of real-world entities is a process that most naturally follows the circumscription of attributes and transformations, and that the attempt to describe a real-world entity via generic properties is not in general a good way to capture its specific characteristics.

These observations motivate an agenda that is broadly consistent with the original aims of object-oriented modelling, but suggest a shift of emphasis, whereby the integrity of objects is given lower priority and more attention is paid to the empirical framework that informs object conception and design. We believe that such an agenda can be addressed within our modelling paradigm, but only by much more sophisticated analysis of the protocols that are implicit in the agent framework that surrounds our scripts. Many of the relevant issues have been more fully addressed elsewhere, in particular, in our proposals for concurrent engineering environments [1] based on empirical modelling principles. The key concept is that of restricting the privileges and patterns of interaction between agents to guarantee the integrity of particular subsets of

observables. This theme is evident in connection with many topics incidentally discussed in this paper. These include: the interaction between the definitions of the time variables in DoNaLD and EDEN alluded to in our case-study, the problematic dependencies between DoNaLD and ARCA components of the jigsaw model, and the tension between introducing generic definitions for entities (such as jigsaw pieces) and faithfully reflecting the particular qualities of each individual entity.

7. Conclusion

The theme of this paper is broad and has connections with many different areas of modern computing. Non-linguistic representations are acquiring an ever more significant role in computer applications (cf the current level of interest in multimedia and virtual reality) and the proper exploitation of graphical representations is a key study of central importance. Much of our work on empirical modelling has already been focussed on broader issues such as agent-oriented modelling in relation to concurrent systems requirements and concurrent engineering and how graphical conceptual representations can be applied in scientific visualisation [1,2,3]. In the longer term, we believe that our approach can be made more effective by developing explicit agent-oriented protocols that in particular will enable closer integration with an object-oriented modelling paradigm.

Acknowledgements

We acknowledge the support of the EPSRC under Grant GR/J13458.
We are grateful to Dr. Steve Russ for his helpful comments. Dr. Beynon is indebted to his mother for an early introduction to jigsaws.

References

[1] V.D. Adzhiev, W.M. Beynon, A.J. Cartwright, Y.P. Yung. A Computational Model for Multi-agent Interaction in Concurrent Engineering, Proc. CEEDA'94, Bournemouth University, 1994, 227-232

[2] V.D. Adzhiev, W.M. Beynon, A.A. Pasko. *Interactive Geometic Modelling Based on R-Functions*, Proc. CSG'94: Set-Theoretic Solid Modelling: Techniques and Applications, Winchester, Information Geometers, 1994, 253-272

[3] W.M. Beynon, M.S. Joy. *Computer Programming for Noughts-and-Crosses: New Frontiers*, Proc. PPIG'94, Open University, January 1994

[4] G. Birtwistle, O-J. Dahl, B. Myhrhaug, K. Nygaard. Simula Begin, Chartwell-Bratt, 1980

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

[6] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns – Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995

[7] D. Harel. *Biting the Silver Bullet: towards a brighter future for Software Development* IEEE Computer, 1992, 8-20

[8] W. Harrison, H. Ossher. *Subject-oriented Programming (A Critique of Pure Objects).* OOPSLA'93, 411-428

[9] S. Parry-Barwick, A. Bowyer. *Woodwark's Method of Feature Recognition*, School of Mechanical Engineering Technical Report 099/1992, University of Bath, December 1992

[10] Wolfgang Pree. Design Patterns for Object-Oriented Software Development, Addison-Wesley/ACM Press, 1994

[11] B. Meyer. Object-Oriented Software Construction, Prentice-Hall International, 1988

[12] M. Minsky. The Society of Mind, Picador, London 1988

[13] A.M. Turing. *Proposal for Development in the Mathematics Division of an Automatic Computing Engine (ACE)*, in Mechanical Intelligence, ed. D.C. Ince, North-Holland, 1992, 1-86