Research Report 329

Enabling Technologies for Empirical Modelling in Graphics

**James Allderidge, Meurig Beynon,
Richard Cartwright, Yun Pui Yung**

RR329

Empirical Modelling is an approach to computer-based modelling that has been under development at the University of Warwick for over ten years. It combines agent-oriented modelling with state representation based on scripts of definitions that capture the dependencies between observables. Unlike conventional modelling methods, its focus is upon using the computer as a physical artefact and modelling instrument to represent speculative and provisional knowledge. Previous research has indicated the potential for wide application to interactive graphics, computer-aided design, concurrent engineering and concurrent systems modelling. This has been demonstrated in principle by numerous case-studies, but existing tools do not yet exploit the underlying concept fully and efficiently. This paper discusses Empirical Modelling in relation to conventional computer programming, describes technical progress towards better practical implementations and identifies new computer architectures on which the aspirations of Empirical Modelling can be realised.

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

July 1997

Enabling Technologies for Empirical Modelling in Graphics

James Allderidge, Meurig Beynon, Richard Cartwright, Yun Pui Yung

Department of Computer Science, University of Warwick, Coventry CV4 7AL

Abstract

Empirical Modelling is an approach to computer-based modelling that has been under development at the University of Warwick for over ten years. It combines agent-oriented modelling with state representation based on scripts of definitions that capture the dependencies between observables. Unlike conventional modelling methods, its focus is upon using the computer as a physical artefact and modelling instrument to represent speculative and provisional knowledge. Previous research has indicated the potential for wide application to interactive graphics, computer-aided design, concurrent engineering and concurrent systems modelling. This has been demonstrated in principle by numerous case-studies, but existing tools do not yet exploit the underlying concept fully and efficiently. This paper discusses Empirical Modelling in relation to conventional computer programming, describes technical progress towards better practical implementations and identifies new computer architectures on which the aspirations of Empirical Modelling can be realised.

1. Introduction

Empirical Modelling (EM) is distinguished from conventional computer-based modelling techniques by its emphasis on the phenomenological aspects of modelling. That is to say, its primary focus is on interpreting the behaviour of interacting agents with reference to the cues by which each agent is responsive to its environment, and the handles by which it changes the state of its environment. Using the computer to assist the analysis of system behaviour in these terms requires a different kind of computational framework. To this end, the emphasis must be put upon using the computer (together with computer-related technology) to construct artefacts, rather than on calculation over abstract computational models.

EM tools and methods so far developed have been successfully applied to a wide-range of interactive programming tasks. These include a model for guest house management, a graphical interface for the construction of statecharts, a model of classroom interaction and numerous simulations, such as a vehicle cruise control controller and an environment for billiards (see *http://www.dcs.warwick.ac.uk/pub/research/modelling/index.html*). EM methods are distinguished by the essential character and nature of the artefacts they generate, rather than by any particular functionality that can be derived from them. EM artefacts can be seen as environments ripe for exploration and experiment, embodying speculative knowledge, subject to empirical confirmation.

In so far as it is appropriate to interpret EM as a programming method, it relates to a preliminary stage of program development, prior to object-oriented analysis and programming. This paper describes some experiments that have been done to explore the relationship between EM artefacts and conventional programs. It also reports on preliminary work to exploit a low-level Virtual Machine as an appropriate target architecture for EM activity. Our investigation motivates further study of novel kinds of computer architecture in which definitive state representations can be combined with traditional object-oriented abstractions, procedural elements and event-driven programming techniques.

1.1 Background to Empirical Modelling

The central concepts behind EM are: *definitive (definition-based) representations of state*, and *agent-oriented analysis and representation of state-transitions*. In broad terms, changes of state within a system (the Referent) are interpreted with reference to a family of observables through which the interactions of agents in the system are mediated. The term **observable** is used in a very general sense to refer to any conceptual entity that an agent can be construed as apprehending in a direct manner. For a human agent, observables comprise not only what can be sensed directly, but what can be inferred immediately, possibly through the acquisition of particular skills or adoption of particular conventions. In modelling other agents in a system, the appropriate observables typically include measurable quantities that cannot be directly perceived by a human agent. The primary aim of the EM process is to construct a computer-based model (the Artefact) in which, by the use of suitable metaphors for interaction and visualisation, the modeller can comprehend the behaviour of agents within the Referent.

Some characteristic principles guide the construction of the Artefact. The behaviour of the Artefact need not be comprehensively understood at any stage during its development. At any given point in the construction process, the modeller has access to a particular state of the Artefact, and has quite general privileges to change this state in the manner of an experimenter. The objective of the experimental process is to determine how faithfully the system being modelling is represented by the Artefact in its current form. Assessing the degree of faithfulness of the Artefact to the Referent is a matter of pragmatic and empirical judgement. If the modeller is satisfied that some aspects of the behaviour of the Referent are sufficiently well-represented by a particular choice of agents, together with associated observables and protocols for their interaction, it may be possible to automate the execution of protocols accordingly. In the contexts for which EM has been primarily conceived, circumscription of agent interaction that leads to full automation is impossible, so that the evolution of the Artefact, whether in respect of enhancement or exploration, essentially involves intervention by the modeller in the role of super-user.

1.2 The Role of Graphics in Empirical Modelling

Computer graphics is closely bound up with EM principles. Visual metaphors play a fundamental role in the representation of agent interaction. Each interface between agents is represented by a family of definitions of variables, or *definitive script*. Each variable corresponds to an observable in the referent, and typically has a value that can be directly displayed. For instance, the principal modelling tool we have developed (the *tkeden interpreter*) supports definitive scripts in which the variables represent a variety of two-dimensional graphical elements, including shapes, point and lines, text strings, windows and displays. The definitions in such a script record indivisible relationships between observables that are empirically identified through interaction with the Referent; these dependencies have a similar quality to the definitions of cells in a spreadsheet. Graphical images constructed by EM in this fashion embody geometric relationships that reflect their semantics and interpretation. In effect, interactive computer graphics plays a significant role in EM, whilst EM principles can be effectively applied to the construction of graphical images.

The use of EM principles at a high-level of abstraction in the process of geometric design is a well-established theme that has been the focus of many previous papers [6]. The idea that such principles are relevant to practical implementation of graphics is a more controversial suggestion, but one that has long been of interest in our research programme [2, 6]. The concept behind the application of EM principles to computer programming is that the ingredients of the computing environment itself can be usefully regarded as a system of interacting agents. In implementing a complex system, these might comprise human agents (such as designers and users), computer processing elements (such as the central processor or processors, and the window manager) and peripheral drivers (such as the screen display, and multi-media IO devices). As the complexity of the reactive system associated with this architecture increases, it becomes ever more

essential to gain conceptual control over the development process. Our research on concurrent systems modelling and simulation suggest that this is a promising area for the application of EM.

1.3 Empirical Modelling and Programming Paradigms

EM artefacts differ from conventional computer programs in their nature, and in their method of use and development. They have a role in the earliest stages of software development, prior to the precise identification and prescription of patterns of interaction between human and computer agents in a system. Our experience has shown that EM is well-suited to the construction of simulations in a wide range of applications, but the use of definitive variables gives a distinctive character to the underlying computational model.

Since definitive variables correspond to observables, they have a status quite different from traditional programming variables. The actions of agents in EM artefacts are developed with explicit reference to the environmental cues and handles through which they interact. The modeller deliberately incorporates these as observables in the model, on the basis that they are needed to account for the observed behaviour of the system. The quality of the Artefact depends crucially on how the modeller construes the behaviour of the system, and the Artefact will typically evolve as the modeller gains deeper insight into the referent through experiment and exploration. Traditional programming methods have been developed with an entirely different scenario in mind: the execution of sequences of primitive operations and interactions whose implications are entirely preconceived in an environment that has already been fully explored.

A useful analogy may be made with the development of a sport such as pole-vaulting. In the first stages of development, both the human performer and the apparatus are involved in an evolutionary process. The performer will be acutely (and perhaps self-consciously) concerned with what sensory cues contribute to successful vaulting, and with what physical characteristics and disposition of the apparatus is appropriate. At some stage, the precise characteristics of the apparatus will have been empirically determined, and the skills required of the performer to some degree systematised. In the final stages of development, the environment presented to the pole-vaulter will be rigorously set up according to standard rules, and the movements of the world-class performer will be performed unselfconsciously according to acquired knowledge with the most economical reference to sensory cues. In this analogy, EM is primarily associated with the first stage, and conventional computer programming with the last.

Failure to appreciate this crucial difference in orientation between EM and conventional programming leads to most inappropriate expectations. For instance, good programming practice in an object-oriented idiom encourages the programmer to look for:

- restrictions that can be placed upon access to variables;
- means to guarantee effective modularisation;
- ways of guaranteeing the absolute integrity of relationships;
- scope for generalisation and optimisation.

In stark contrast, the computational qualities of definitive variables reflect the empirical status of the knowledge about observables that is expressed in the Artefact. Without presuming more than mere empirical insight guarantees, it is impossible to be sure whether there are unrecognised dependencies and interactions between observables (as subtle as those proposed by astrologers, for example). It is likewise impossible to be sure that relationships reliably observed in the past will persist in the future, that the environmental response to the actions of an agent is so consistent as to permit optimisation, and that what appear to be several instances of one class of agents are genuinely similar.

Of course, in practice, the EM process serves to identify valid protocols for interaction with definitive variables. Each artefact can be used as a source of environments, established by selecting appropriate

definitions and agents, that can be used in 'what if?' experiments to clarify and refine the empirical knowledge that underpins a conventional implementation. A central objective of such experiment is to identify where restrictions can be placed upon access to variables, where modular boundaries can be conveniently established, where constraints upon relationships between observables can be reliably presumed, and where optimisation and generalisation is possible. In these respects, EM is complementary and prior to the application of object-oriented methods.

EM artefacts have in addition distinctive qualities of their own. The use of definitive representations of state is a powerful means of simplifying the description of complex models, and of representing synchronisation between agents. In principle, it can be used in conjunction with object-oriented abstractions to redress such problems as the conflict between strict encapsulation and the need for global information, to supply relations among objects, and to assist constraint management. The power invested in the modeller as super-agent introduces openness to the modelling process, and freedom from premature circumscription.

1.4 Empirical Modelling and Knowledge Representation

The essential quality of EM artefacts is the integration of two kinds of knowledge: *speculative* knowledge, associated with empirical insights that are yet to be fully explored and confirmed as valid, and *assured* knowledge about agency and the environments in which they interact. Conventional programming largely relies upon assured knowledge about the environment for interaction between human and computer agents. Only with such knowledge is it possible to predict the operational interpretation of a computer program. A formal specification of a program can only capture those aspects of the interaction between the human and the computer agents in a system that lie within a preconceived closed world of reliable primitive operations and communications.

The limitations of formal programming methods in relation to graphics are generally recognised; they reflect the fundamental impact of empirical concerns. Consider, for instance, the aesthetic qualities of a graphical display, the psychological effect of different strategies for screen update, and the sensory cues by which a particular choice of texture or mode of presentation enhances realism, or conveys meaning. In such matters, experimentation with a model is essential, and the importance of devising the graphical model so as to enable such experimentation is paramount. EM addresses such concerns, emphasising the use of phenomenological variables to represent observables, representing interaction between agents in terms of privileges to observe and redefine observables, and interpreting system behaviour with reference to indivisible patterns of simultaneous change of observables. Speculative knowledge that falls short of comprehensive understanding and circumscription of the interaction between agents in a system is captured in a computer-based artefact, on which the modeller can perform experiments to evaluate empirical insights.

1.5 Modelling the Application and the Machine

The idea of using computer-related technology to construct artefacts is a radical departure from that of the computer as a generalised calculator. The historical roots of conventional programming lie in the classical theory of computation, with its emphasis on formal recipes of operations on a reliable machine. For this reason, conventional programming paradigms are not well-oriented towards treating the computer as a physical object. Abstract data types and objects can assist the task of bringing computational abstractions into line with the programmer's conceptual grasp of an application to some degree, but deal with concepts that presume assured knowledge, and do not explicitly address the way in which such structures have to be apprehended by the programmer. For instance, object-oriented modelling ensures the local consistency of state within object boundaries, but does not pay comparable attention to "what is perceived as indivisible".

EM aims to establish a more direct relationship between application and machine, without the artificial

intermediate constraints that a conventional computer programming perspective introduces. It emphasises the machine as a physical object, like the engineering prototype, and the modelling process as an empirical activity focussed on a phenomena and a physical model. Unlike conventional program variables, definitive variables have an explicit phenomenological significance; they are intended to be experienced. The kinds of physical model of abstract entities demanded by different kinds of application are diverse, and the effective use of a computer to construct such models is an essentially empirical process, similar in character to the development and use of an instrument. For this reason, EM is directed both at faithfully imitating the application and at developing the machine as a modelling instrument.

2. Empirical Modelling in Practical Implementation

2.1 From Empirical Modelling to Conventional Implementation

The above discussion motivates the investigation of EM as a framework within which programs or even systems for graphics can be developed. One possibility is that of constructing EM artefacts that can be used as a source of conventional programs. In effect, the modeller first constructs a general-purpose model of an application that admits uncircumscribed open-ended interaction, then derives conventional programs from this model by an automatic or semi-automatic translation process. The input for the translator comprises the model, together with agent specifications for the intended users. Where there is a change of requirement, it is then possible to return to the model, and make a new translation using different agent specifications, possibly after first modifying the model.

The resulting program will typically have a much more restricted and focussed functionality. It may also be customised for a particular machine architecture, whether abstract or actual. For instance, if the target is implementation in an object-oriented language, the EM process will have to be appropriately directed towards the identification of generic abstractions. (See [1] for a discussion of the role that higher-order dependencies can play in this respect.) The choice of visualisation method in the EM process should likewise reflect the display mechanisms available on the target architecture.

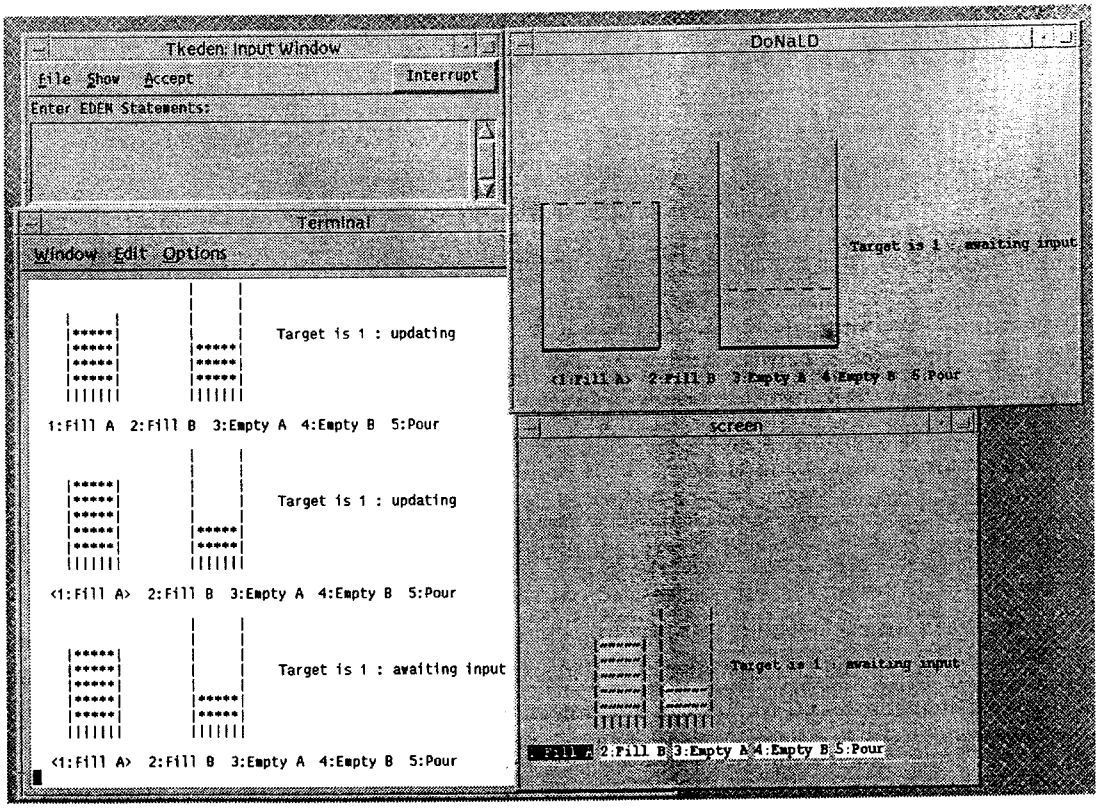

Figure 1

By way of illustration, Figure 1 depicts an EM model associated with a simple program that was first developed for educational use in schools (cf [4]). The intended functionality of the program is clear from the menu buttons: the objective for the pupil is to realise the specified target quantity of liquid in a jug by appropriately filling and emptying jugs and pouring from one jug to another. The model includes a core system of definitions that maintains relationships between the key parameters in the application: in this context, the capacities and contents of the jugs, and the availability status of the menus. The use of definitive scripts make it possible to directly express the relationship between menu availability and key parameters in a simple and direct manner, thus:

contentA = ... **definitions**
capA = ...
fullA = *contentA*==*capA*
avail_option_fillA = **not** *fullA*
colour_button_fillA = **if** *avail_option_fillA* **then** *white* **else** *black*

The modeller can interact with the jugs model in the role of a super-user, for instance, by directly redefining the values of key parameters. For instance, the redefinition

contentA = *target* **super-agent action**

can be used to simulate successful completion of the user's task. Such a mode of interaction is not within the scope of the functionality of the translated program to be derived from the model; for this purpose, the valid interactions are defined by a choice of input menu, represented in terms of our definitive representation of state by a redefinition of a variable *input* (=*fillA, fillB, emptyA, emptyB, pour*), as in:

if *avail_option_fill* **then** *input* = *pour* **user interaction**

Another agent within the jugs model controls the pouring mechanism. A simple extract from the protocol for this agent is:

event-driven action
if *input*==*pour* and **not** *emptyA* **and not** *fullB* **then**
{*input*=*pourAB*; *contentB* = |*contentA*+*contentB*|-*contentA*}
if *input*==*pourAB* **and not** *emptyA* **and not** *fullB* **then** *contentA* = |*contentA*|-1

As the definitions listed above illustrate, the scope of the observables associated with the jugs model encompasses characteristics of the device on which the model is being constructed. As Figure 1 illustrates, the precise form of the visualisation in the jugs model can be based on quite different display capabilities and generic ways of constructing patterns. In the context of Figure 1, the three different visualisations are specified using three different definitive scripts simultaneously, one based on a definitive notation for line drawing, one on a definitive notation for window layout, and one on textual output.

2.2 A Spectrum of Translation Strategies

The jugs model illustrates the principal concepts represented in an EM artefact: the definitions that specify the model of the application and the characteristics of the computer artefact; redefinitions (only available to the super-agent) whose role is confined to the experimental process involved in the construction of the model; event-driven actions that perform the functions of circumscribed agents internal to the model; procedural actions that are invoked in a particular mode of use of the model.

One motivation for translating such a model into a conventional program to realise a particular function is that greater efficiency of execution can be obtained in this way. In effect, translation achieves efficiency at

the cost of flexibility, restricting the mode of interaction with the model in order to be able to optimise the response. The translation process also gives insight into the prospects for parallel execution. Figure 2 illustrates a spectrum of translation strategies that can be used.

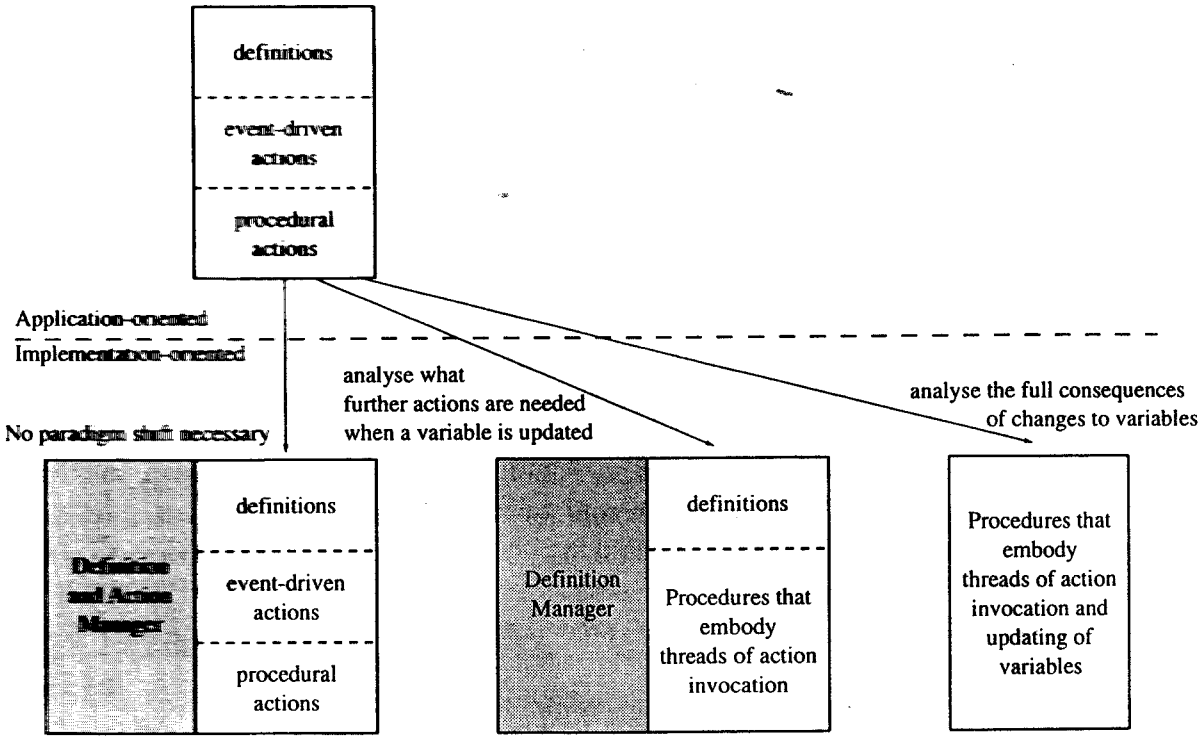

Figure 2

The application-oriented EM artefact is represented by the single box at the top of the diagram. The boxes underneath are associated with particular types of implementation in general purpose languages. Possible target languages for translation include C, C++, Java, and the specially devised languages DAM (Definitive Assembler Maintainer) and JaM (Java Maintainer) to be discussed further below. The shaded components in Figure 2 are application-independent; they perform general functions of dependency maintenance associated with processing definitive scripts. This involves keeping the values of internal variables (cf. *contentA* an *capA* in the jugs model) up-to-date, and ensuring that their value is consistent with their visualisation (for instance, by invoking procedural actions to update the screen display).

Where a definition and action manager exists in the target environment, no change of paradigm is involved in the translation process. The translation in this case resembles a change of dialect, analogous to translation from one European language to another. Moving from left to right in the spectrum of translation strategies in Figure 2 entails a change of paradigm, analogous to translation from a European language to an Eastern language.

Many different approaches to translation have been investigated. A brief survey of some of the strategies investigated follows.

2.3 Direct conversion into a Procedural Program

Methods of this nature entail analysing the implications of all changes to variables, taking account of propagations of change associated with dependency maintenance, and of any actions triggered by consequent changes of state. In this approach, the analysis of the application that is required can be very complex, and it is difficult, but not impossible, to account for actions that are represented by 'genuine' redefinitions of variables (as opposed to assignment of a new explicit value to a variable). For instance, the

translation of the jugs program is complicated by the introduction of a dependency between *contentB* and *contentA* when pouring (see the event-driven action associated with selection of the *pour* menu above).

An alternative strategy that has been examined is the introduction of a definition class, together with potential for action management, to an object-oriented language such as C++. By this means, the definitions in the application model can be directly represented by instances of definitions, and procedural code can be generated by conventional techniques.

2.4 Translation in the Context of a Definition Manager

Two different but related approaches have been taken to the implementation of a definition manager. The first is the development of DAM; the second, the development of JaM. Both DAM and JaM are essentially concerned with the maintenance of definitive scripts. JaM allows definitive scripts to be defined over extensions of specific Java classes. In contrast, DAM operates at a low-level of abstraction. For instance, the DAM virtual machine enables the programmer to establish definitive relationships between words of RAM store. In translating an EM artefact to DAM, it is then possible to convert dependencies expressed in high-level definitive scripts into equivalent scripts that establish relationships between machine words. This process (to be illustrated below) is analogous to the way in which models of dependency can be refined without changing their essential character by invoking observation at a new level of detail. For instance, in modelling a linkage the overall operation is the same whether or not the dependencies between its internal components are modelled. Both DAM and JaM can potentially be extended to handle actions.

2.5 Translation in the Context of a Definition and Action Manager

Perhaps the simplest approach to implementation is to implement the full functionality of a Definition and Action Manager in the target language. For instance, the Eden interpreter is itself implemented in C, and can be wrapped in such a way that it can be included in and called from within a C program. In this context, it is possible to develop direct C equivalents for the definitions, event-driven actions and procedural actions in the EM artefact. The potential advantage of this method over direct use of the Eden interpreter is that it allows compilation of a program derived by restricting the functionality of an EM artefact.

Other variations on this theme include partial translation of an EM artefact into procedural terms. For instance, in a billiards simulation we have developed, the path of a ball is first computed in an environment in which the possibility of unexpected intervention by the player is permitted (the player may for instance opt to hit the ball twice, or move the object ball just prior to impact). Openness to intervention of this nature introduces computational overheads that detract from the smoothness of the animation. The degree of realism in the motion of the ball is also influenced by other dependencies in the environment - for instance, impacts of balls are monitored to compute and update the score as necessary. By judiciously suppressing peripheral agency and dependency, and retaining only such essential dependencies as are associated with visualisation of the ball, it becomes possible to optimise the animation, and so replay a shot more realistically.

3. Efficient Implementation Strategies

3.1 Parallelisation

When translating from an EM artefact, one virtue of retaining a Definition and Action Manager is that some potential for parallelisation is apparent in this aspect of the computational task. Definition and action management essentially comprises three phases:

1. mark out-of-date variables i.e.
   - determine which variables need to be updated
   - determine what actions should be fired
2. update the out-of-date variables
3. execute the actions.

Since some variables may be changed by the actions, it is necessary to cycle through phases 1 to 3.

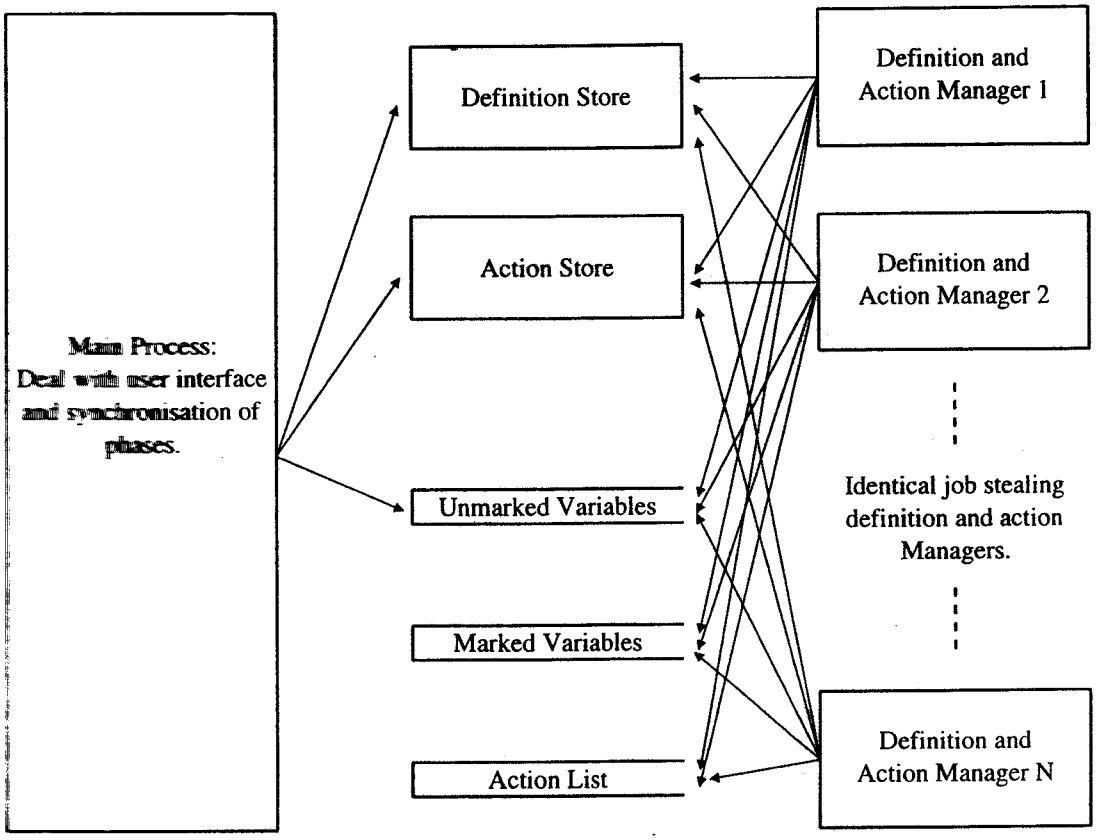

Figure 3

Each phase of this process can be parallelised. The overhead of setting up threads may be too costly, but a more plausible way is to implement the parallelism using shared memory. As depicted in Figure 3, a main process is responsible for user interfacing, and setting up the Definition Store and Action Store. A user's request to redefine of a variable will set up the variable list that records the variables to be marked. As indicated diagrammatically in Figure 3, several Definition and Action Managers will mark variables, update variables and perform actions in parallel, following a work stealing strategy. To this end, an idle Definition and Action Manager will consult the variable and action lists in order, observing the rules that no variable should be updated if there are variables to be marked, and no action should be performed if there are variables to be updated. The computational activity follows the pattern indicated in Figure 3, where the lists of Unmarked Variables, Marked Variables and Actions change dynamically until all three lists are empty. At this point in the computational cycle, the main process responds to the next user-initiated request to modify the system state.

3.2 Definitive Assembler Maintenance

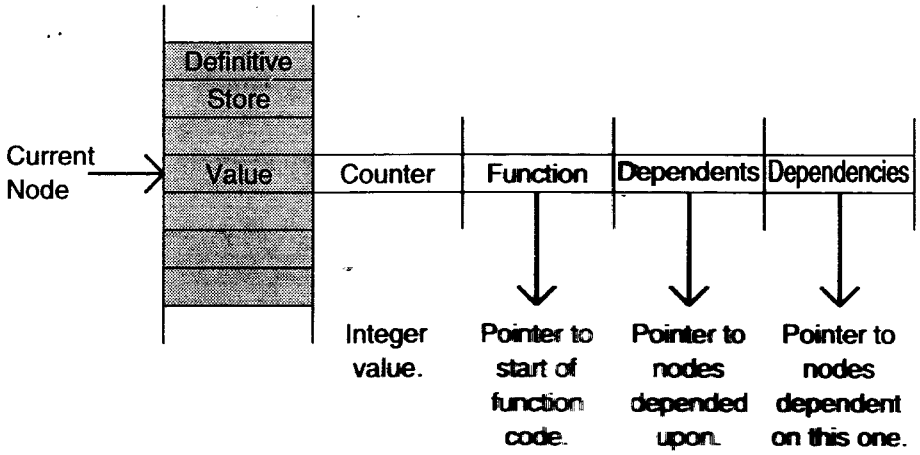

Figure 4

The Definitive Assembler Maintenance (DAM) machine is a low-level Definition Manager that maintains dependencies between words in RAM store. The current implementation, due to Richard Cartwright, uses the 32-bit architecture of the ARM chip [5]. As indicated in Figure 4, the DAM builds an internal dependency data structure to represent indivisible relations between 32-bit words. Each function that can appear on the right hand side of a definition is user-defined according to the convention that it takes its arguments from the registers r0, r1, r2, ..., r9 and return its result to r0. In this fashion, it is possible to maintain dependencies between any data items that can be represented in 32 bits, such as integers, floats, booleans and pointers. Since one such data item can in principle be a machine instruction, the possibility of self-modifying code is also admitted.

The maintenance of dependency is handled by a small piece of ARM code. This uses Knuth's algorithm for topological sorting to update dependent variables in an efficient and discriminating manner, and accesses the counter in the dependency data structure for this purpose. The other elements in the dependency data structure are pointers to the list of words whose values depend upon the current word, and to the list of words upon which its value depends. Since one conceptual data item (such as line on the screen) has to be represented by several words in DAM store, it is essential to have a means of executing several definitions or redefinitions 'simultaneously'. This can be achieved by storing update requests prior to execution.

In practical use, access to the DAM machine is via an Application Programmer's Interface. Via this interface, the programmer can decide the memory layout, and can read the contents of memory locations. The contents of locations can only be overwritten by notifying DAM of a redefinition. As indicated in Figure 2, DAM serves as a virtual machine that plays the role of a low-level Definition Manager. The possibility of directly connecting the DAM store with the screen memory has been investigated as a plausible route to a purely definitive implementation paradigm, but it appears to be more efficient in practice to implement screen updates via calls to the operating system. The dependency structure in DAM can nevertheless be useful in this respect, since it makes it possible to introduce variables whose function is to provide cues for screen updates when display elements have been modified.

3.5 A DoNaLD to DAM Compiler

The tkeden interpreter incorporates three definitive notations: DoNaLD, a definitive notation for line drawing, SCOUT, a definitive notation for screen layout, and the definitive component EDEN, an evaluator for definitive notations that allows definitive scripts to be formulated over scalar types, non-homogeneous recursive lists and strings. The DAM architecture was conceived as an appropriate platform on which to implement tkeden, since scripts in each of these definitive notations could in principle be represented

directly on the DAM store. DoNaLD is the first notation to have been implemented in this fashion, using a compiler that was developed by James Allderidge as a final year undergraduate project.

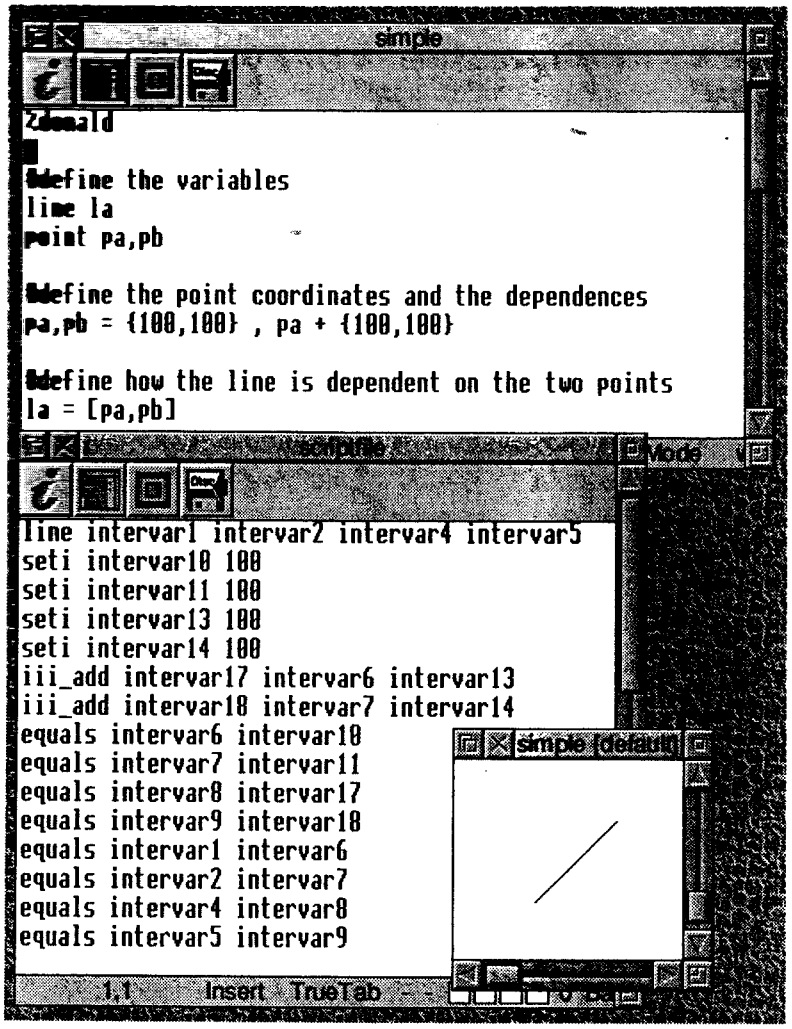

Figure 5

There are two phases to the compiler. In the initial *parsing* phase, a DoNaLD script is converted into a family of abstract definitions with associated type information that is then transformed into a symbolic DAM script by factorising the abstract variables into word-size components. In this way, a DoNaLD line is represented by four DAM words, as in Figure 5. In the second *code generation* phase of the compilation process, the symbolic DAM script is realised as a sophisticated data dependency structure in DAM store.

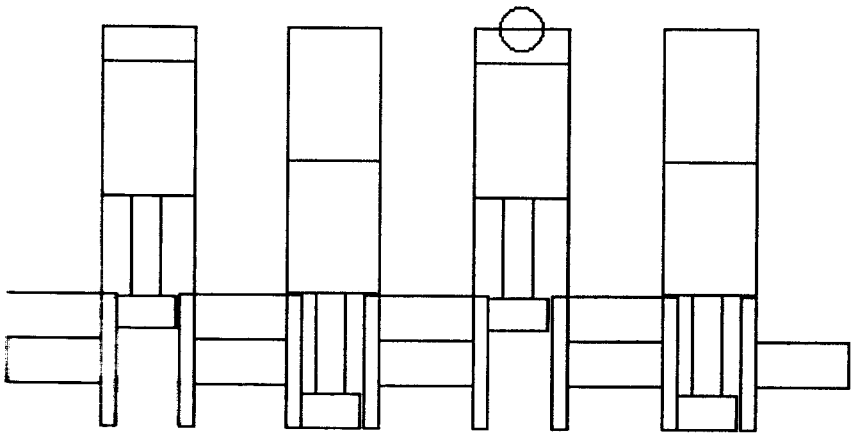

Figure 6

In its present form, interaction with the DoNaLD script is restricted so that the only possible redefinitions are to scalars at the leaves of dependency trees. This degree of interaction is sufficient to allow us to assess the efficiency of the updating of geometric dependencies in animation. For instance, Figure 6 depicts a section of a four cylinder engine, parametrised in such a way that the firing cycle can be animated. The DoNaLD script has over 70 definitions, and translates to a DAM dependency structure with 890 entries. This animation can be executed smoothly at 25 frames per second on a modest personal computer. The speed of this animation surpasses what is typically achieved using the tkeden interpreter by a factor of ten.

4. Further Work

The DoNALD to DAM compiler demonstrates the principles of implementing a high-level definitive notation using a low-level Definition Manager. Implementing all the definitive notations incorporated in the tkeden interpreter is a obvious next step. At present, the interaction with the compiled script is much more restricted than the DoNaLD interpreter allows. Possible extensions include scope for redefining the values of explicitly defined variables of non-scalar type, for introducing new variables, and for redefinition of variables using new defining formulae. The DAM architecture also has potential of a complementary kind: since intelligible access to its encoded variables relies upon an explicit interface, it becomes possible to make DAM code public, yet restrict the functionality that is accessible to particular users.

Previous work on EM has drawn attention to the need at a high-level of abstraction for dependency relationships that govern both references to observables and the privileges of agents to interact with observables [3]. One motivation for investigating the DAM architecture is that dependencies between machine words can serve a variety of semantic functions, according to whether a word is interpreted as a value, an address or an instruction. The DoNaLD to DAM compiler makes only very limited use of dependency-linked addressing (for instance, it has a role in implementing conditional definitions of string messages), but it is possible to envisage more sophisticated applications, in particular in connection with higher-order definitions such as were discussed in [1].

Preliminary research on JaM, which is being used for the implementation of the definitive notation for geometric modelling CADNO [6], indicates that one of the advantages of having a definition manager in the abstract machine level is that it in principle can give more support to the concept of machine as modelling instrument. For instance, JaM redefinitions can be used not only to manipulate geometric objects, but to change rendering attributes and viewing parameters in a flexible manner. The implementation of high-level definitive notations for line drawing and screen layout in DAM may offer similar advantages.

References

[1] D Gehring, Y P Yung, R I Cartwright, W M Beynon, A J Cartwright *Higher-order Constructs for Interactive Graphics in a Definitive Programming Framework* Proc. Eurographics UK '96, Vol. 1 1996, 179-192

[2] W M Beynon *Evaluating Definitive Principles for Interactive Graphics* New Advances in Computer Graphics, Springer-Verlag 1989, 291-303 - Computer Science Research Report 133, University of Warwick 1988

[3] 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

[4] W M Beynon, M T Norris, S B Russ, M D Slade, Y P Yung, Y W Yung *Software Construction using Definitions: an Illustrative Example* Computer Science Research Report 147, University of Warwick 1989

[5] P Cockerell *ARM Assembly Language Programming* Hemel Hempstead, M.T.C., 1987

[6] W M Beynon, A J Cartwright *A Definitive Programming Approach to the Implementation of CAD Software* Intelligent CAD Systems II: Implementation Issues, Springer-Verlag 1989, 126-145