Chapter 5

The DAM Machine

5.1 Introduction

The DM Model presented in Chapter 4 is a model for dependency maintenance for representing dependencies between one data type $\mathcal{Z}$. This chapter considers the question of whether it is possible to implement the DM Model, with its potentially infinite length value representation, on a computer system using multiple finite length integers. Motivated by the fact that computer systems use memory store that is made up words containing binary bits, with state either on or off, to represent all data of all types at the lowest level of representation, it seems appropriate that a n-bit word can be considered analogous to the data type $\mathcal{Z}$ in an implementation of the DM Model. This chapter presents the design for a dependency maintenance tool written in assembly code that maintains indivisible relationships between words in an area of computer RAM store. The tool is the called the *Definitive Assembly Maintainer Machine* (DAM Machine) and the area of words in store maintained consistent with its definition is called the *definitive store*.

The first section of this chapter (Section 5.2) explains the way in which the DM Model is implemented by the DAM Machine. The relationship between the

mathematical model and the implementation is explained by relating the layout of store in the DAM implementation to the mappings and sets of the DM Model. The implementation uses the block redefinition algorithm[1] to keep the values in store consistent with their definitions. The DAM Machine is accessed through programming language function calls allowing a programmer to use an area of definitive store in their applications. The process for creating and accessing the store is explained.

The implementation design requires the programmer of a definition based model to decide how to represent observables as words in definitive store memory. It is necessary to consider the location in memory of a word and how the pattern of bits is significant to a higher-level data type, in the manner required for standard low-level assembly coding. The dependency maintenance in a script model is taken as close to the processor as possible in DAM Machine implementation, creating an efficient dependency maintaining virtual machine, without the need of an additional runtime interpreter or translator. Issues of data representation in the DAM Machine are discussed in Section 5.3 along with the stages of conversion of an existing definitive script notation into a tool that uses the DAM Machine to maintain its dependency.

A translator of the DoNaLD notation [ABH86], the notation for line drawing integrated into the *Tkeden* package, that uses DAM as its dependency maintainer instead of EDEN is presented in Section 5.4. This was developed by Allderidge as part of his third year undergraduate project work. This implementation uses the same parser as the DoNaLD notation and generates an intermediate script of definitions in a format that is suitable for execution on the DAM Machine. The translation process is explained with reference to the stages of the construction of the DM Model. Efficiency claims for the DAM Machine concept are supported by a comparison of timings between the *Tkeden* tool and a DAM Machine representation

[1]See Section 4.3.3 of Chapter 4.

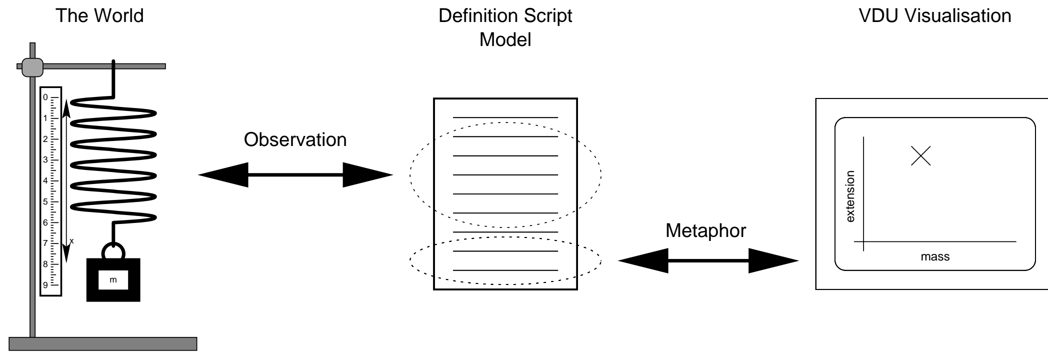


Figure 5.1: Script describing a model and its visual metaphor.

of the same DoNaLD model.

The *Vehicle Cruise Control Simulator* [BBY92], developed to run in *Tkeden*, demonstrates how at any point of interaction with the interpreter of an empirical model, it is possible to either refine the script of definitions of the model or the metaphor for its visualisation. For example, it is possible to redefine the current speed and observe the change to the speedometer or to redefine the geometry of the speedometer at any point through the interaction. Figure 5.1 shows how this idea could be extended to a low-level maintainer such as DAM. The current output of a visual display of a computer system is held in the memory of the computer itself. The location of a set of pixels forming a cross pattern on the screen can represent the observed mass / extension relationship of a spring in an observed model that is represented as a script of definitions. By placing the pixels of the display in an area of memory where words are maintained consistent to their definition, the *definitive store*, it is possible to build a direct link between the model as observed and visualised. An illustration of this process for geometry is presented in the last section of the chapter (Section 5.5).

5.1.1 Scics Machine

An experimental precedent for the DAM machine is the em Scics Machine, prototyped by Simon Yung [Yun96]. Scics combines the UNIX-based spreadsheet "*sc*" with a computational machine simulator. This simulator was developed at the University of Warwick with the aim of assisting with the task of teaching of computer science. Machine code and the machines memory resides in cells of the spreadsheet. Dependency maintenance is carried out by the spreadsheet and this takes precedence over machine execution.

Yung used the *Scics Machine* to implement heapsort [AHU82] in such a way that, for example, spreadsheet dependencies serve to update the heap conditions when values on the heap are exchanged[2].

5.2 From the DM Model to DAM Implementation

A DM Model represents a script of definitions transformed from a high-level as a low-level script with one data type, the integers $\mathcal{Z}$. The DAM Machine is an implementation based on the DM Model where there is also only one data type, a word of computer *Random Access Memory* (RAM) store. A state i of a DM model $\mathcal{M}_S = (A, F, D, V)$ can be related to the state of the DAM Machine memory. Different computer architectures use different lengths of words, each containing n-bits. On an n-bit architecture there are 2^n possible values for each word. The computer system on which the DAM Machine is implemented is the Acorn Risc PC that uses an ARM RISC processor [Fur96] and is a 32-bit architecture. For this architecture, each word has a total of 2^{32} possible values.

The set of references A of the DM model is replaced by a range of addresses in store for the DAM Machine. These locations in memory are known as *definitive*

[2] Beynon recently developed a *Tkeden* interactive model for the heapsort algorithm and this is described in [Bey98b].

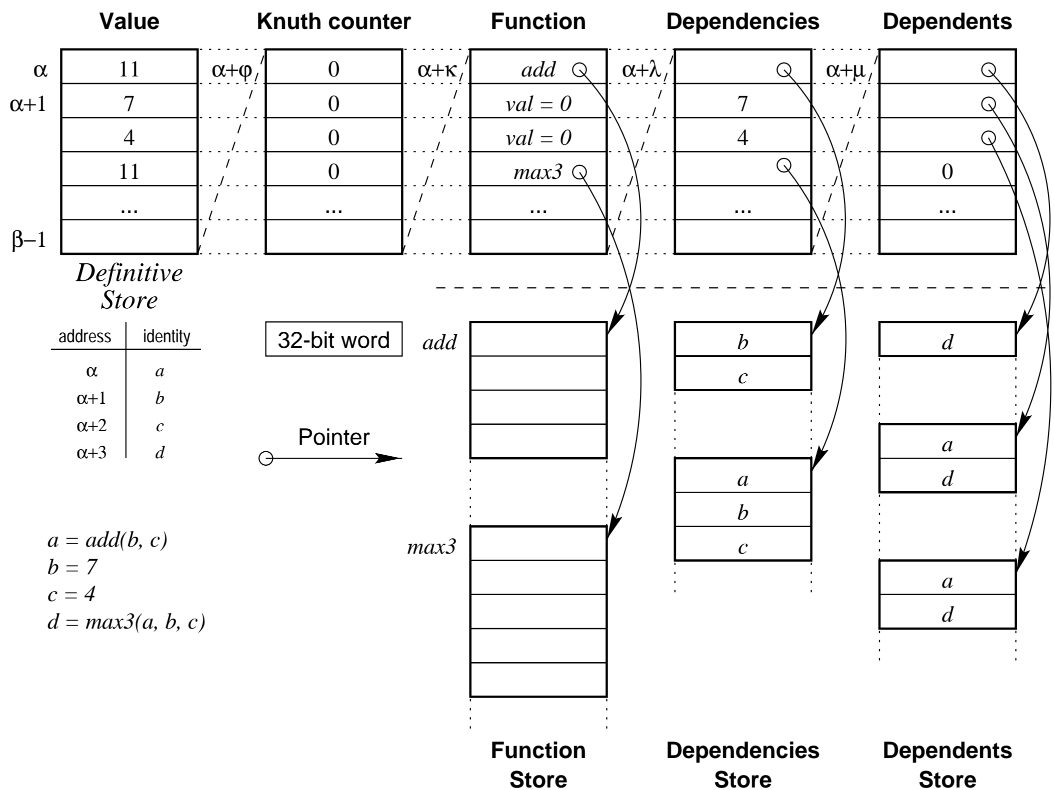


Figure 5.2: Design of the memory layout for the DAM Machine.

store, a continuous section of memory with a range bounded by low address α and high address $\beta - 1$. Unlike the DM Model, the implementation design requires that this set is the same size throughout the life of a script and that all new definitions are actually redefinitions of existing definitions. The initial state of the DAM Machine is that all references are associated with the DAM equivalent of the $value_0$ function, have no dependents and an initial value of 0. Figure 5.2 shows the design of the memory layout for the DAM Machine, with a definitive store starting at address α and ending at address $\beta - 1$ in the block of words in the top left hand corner labelled *Value*.

An example script shown at the bottom left hand corner of Figure 5.2. Above this script is a small table showing how the identifiers in the script are mapped to addresses in the DAM Machine. The rest of the figure is a graphical representation

of the memory layout for the implementation of the DAM Machine based on the example script.

In a DAM Machine with a definitive store of size $s = \beta - \alpha$, additional store is required to represent the dependency between addresses. Four other areas of store exist that are the same size as the definitive store. Each area is used to hold information used to maintain dependencies between the words in definitive store. For each word, these are used to represent:

- pointers to the code for the operator used to maintain an indivisible relationship between the value of the word and the value of other words in the definitive store (*function* in the figure);
- pointers to a list with elements that are locations in store on which the word depends (*dependencies* in the figure);
- pointers to a list with elements that are locations in store that depend on the value of this word (*dependents* in the figure);
- a *Knuth counter* for use in the block redefinition algorithm. See Section 4.3.3 for details of the *Knuth counter* value in the block redefinition algorithm.

For a store of size s, there must be at least five times this number ($5s$) of words of memory to represent the dependencies in the store. An area of heap memory is required to store the lists of pointers (the *dependencies store* and *dependents store* in Figure5.2) and an area of store for the assembly code used to maintain dependencies (the *function store* in the figure).

The block of store for the Knuth Counters in the DAM Machine model are offset from the addresses of definitive store by φ ($\varphi \geq s$), starting at low address $\alpha + \varphi$ and ending at high address $\alpha + \varphi + s - 1$. The current Knuth Counter for a value stored at address $\alpha + j$ is definitive store is itself stored at address $\alpha + \varphi + j$.

The other three blocks of memory are organised in a similar way, with the function block offset by κ, the dependencies block offset by λ and the dependents block offset by μ[3].

The block for function pointers corresponds to the mapping F in the DM Model, which maps any reference to the operator that is used to calculate the value connected with the reference. All mappings in the DM Model are members of the set $\mathcal{F} = \{f \mid f : \mathcal{Z}^* \rightarrow \mathcal{Z}\}$, comprising functions from a sequence of values of data type $\mathcal{Z}$ to a value of type $\mathcal{Z}$. The DAM Machine equivalent is a block of code that calculates the return value of one word from a sequence of words. These mappings can be mathematically represented as members of the set $\{f \mid f : [0, 2^{32} - 1]^* \rightarrow [0, 2^{32} - 1]\}$. More details of DAM operators can be found in Section 5.2.1.

The *dependencies* for each word in definitive store, references to the values that the word depends on that are also the arguments to the associated defining operator, are stored in a block offset by λ from α. For a value in definitive store at address $\alpha + j$, a pointer to a singly linked list for the dependencies for that value is stored at address $\alpha + \lambda + j$[4]. This list, stored in the operating systems heap, contains addresses for the values in definitive store that are dependencies in the same way that the mapping D associates a reference with a sequence of references that are dependencies in the DM Model. If the pointer in the operator block is equal to 0 for a particular associated value in definitive store, the value has no dependencies and it is assumed that its operator is equivalent to the DM Model $value_X$ operator. In this case, X is stored in the associated dependencies block instead of a pointer to a list.

The *dependents* block of store at an offset of μ from α has no analogous

[3]The relationships between φ, κ, λ, μ and s are as follows: $\kappa \geq \varphi + s$, $\lambda \geq \kappa + s$, $\mu \geq \lambda + s$.

[4]In Figure 5.2, the *dependencies store* and *dependents store* are represented as blocks of memory. In implementation, these are singly linked lists, the order of which determines the order of the sequence of arguments to the operator.

component in the DM Model. For a word in definitive store at address $\alpha + j$, a pointer to a singly linked list of the addresses for dependents of that word, those that have the word as a direct dependency, is stored at address $\alpha + \mu + j$. This list is maintained to assist the efficiency of the algorithms that update the store with redefinitions by representing a doubly linked dependency structure. If the value at an address in the block is 0, the associated word in definitive store is a *root node* at the top of a dependency structure for the definitions represented.

5.2.1 DAM Machine Operators for Definitions

A definition in a script prepared for the DM Model is of the form

$$identifier = operator(arguments).$$

In the DAM Machine, an operator is analogous to a block of assembly code that is passed address references to words that represent the *arguments* sequence of a definition, in the current states of processor registers. The code should calculate a value based on looking up the values stored at these addresses and place this calculated value back into a register `R0`. For any definition, the DAM Machine ensures that before control is passed to the block of code for an *operator*, the correct references are stored in the registers to represent the *arguments* and that the value remaining in the register is stored into the address in definitive store for the *identifier*.

For example, consider the definition "*a = add(b, c)*" shown in Figure 5.2. The operator *add* should be a block of assembly code instructions that lookups the values stored at the addresses for b and c (that are $\alpha + 1$ and $\alpha + 2$ respectively), load these into registers, add the values in the two registers together and move the result into register zero. The block of code is passed a pointer of where to return control after execution and should end with an appropriate branch instruction to

do this. The ARM code for an *add* operator is shown below[5].

```
.add LDR R2,[R0]   ; Load R2 with defn. store value at R0 (b)
     LDR R3,[R1]   ; Load R3 with defn. store value at R1 (c)
     ADD R0,R2,R3 ; Set R0 equal to R2 + R3 (b + c)
     MOV PC, R14  ; Return control to the DAM Machine
```

A programmer who is implementing a DAM Machine dependency driven part of an application must write the assembly code for these operators. The code should be consistent with their chosen data representation over the 32-bit words supported by the DAM Machine. More details on the data representation strategy are presented in Section 5.3.

5.2.2 Redefinitions and the Update Mechanism

A programmer who wishes to use a DAM Machine in an application is required to allocate sufficient memory to store the number of definitions they wish to represent in definitive store by choosing appropriate values for α, β, φ, κ, λ and μ. The initial state of the DAM Machine definitive store is that every value is set to 0, every associated Knuth counter is set to 0, every function pointer and dependencies pointer is 0 to represent the special $value_0$ operator and every dependent is 0. They must also place into memory the assembly code for the operators, as described in Section 5.2.1. To redefine a value in store at an address that lies between α and $\beta-1$, the programmer must load into the processor's addresses the parameters shown in the list below and then branch to the "`addtoq`" subroutine of the DAM Machine.

`R0` The address of the value in definitive store to be redefined.

`R1` A pointer to the start of the block of operator code used to calculate the value implicitly defined by the definition for the associated identifier in `R0`. If `0` is in

[5]In the current implementation, the references to addresses that are dependencies are passed to the operator subroutine in registers `R0` up to `R10` and the return pointer for the subroutine in `R14`. This enforces an implementation specific limit on the length of the argument list to a maximum of 11.

this register then the special $value_X$ operator is assumed, in other words the value at the address in `R0` is explicitly defined to be X in register `R2`.

`R2` If `R1` is loaded with a `0` then `R2` is the value of X for the special $value_X$ operator. Otherwise, register `R2` should be loaded with a pointer to the value in definitive store that is the first argument in the definition argument sequence for the redefinition.

`R3` to `R12` The pointers stored in these registers represent the elements with indices 2 up to 11 of the argument sequence for the redefinition. These pointers should be addresses in definitive store that are dependencies for the value stored at the address in `R0`. The first value of `0` in order through the registers signifies the end of the argument sequence and the order of pointers in increasing register index is the same as the order of the sequence for the redefinition.

This causes the DAM Machine to place the redefinition of the queue of redefinitions awaiting that next branch to the DAM Machine "`update`" subroutine. This queue is equivalent to the set K in the DM Model. Figure 5.3 shows two redefinitions for the script used in the example in Figure 5.2, "*d = add(a, c)*" and "*c = 3*". Directly below these redefinitions in the figure are the registers that should be set before the call to the "`addtoq`" subroutine of the DAM Machine. The effect if this subroutine is to store these register values in an area of memory internal to the DAM Machine.

When the "`update`" subroutine is called for a DAM Machine in state i equivalent to a DM Model $\mathcal{M}_S = (A, F, D, V)$, this queue of definitions is considered the set K of redefinitions. The block redefinition algorithm is used to update the state of the DAM Machine to state $i + 1$. If any the redefinitions on the queue introduce cyclic dependency into the model, an error is reported and the previous state i of the DAM Machine and definitive store is restored. Along the bottom section of Fig-

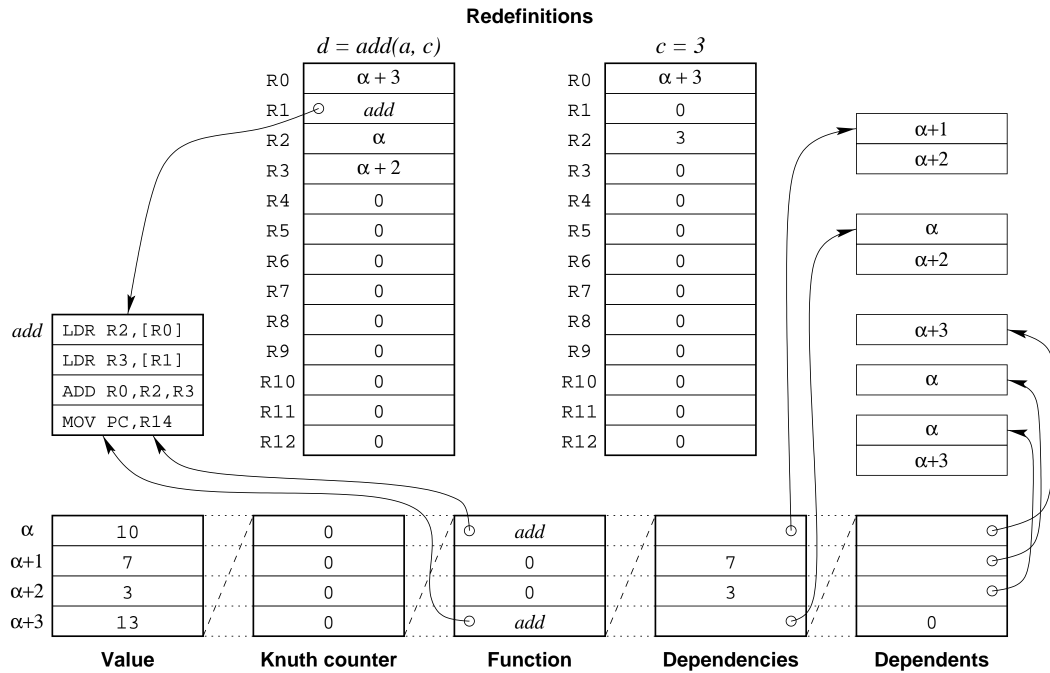


Figure 5.3: Two redefinitions and their effect on DAM Machine Store.

ure 5.3, the successful call to the "`update`" method due to the two definitions results in the definitive store and other associated blocks of store. The lists of dependencies and dependents are shown down the right hand side of the figure and the block of code for the operator is shown on the left hand side.

It is up to the programmer to ensure that they choose a suitable layout for definitive store for their application. If a programmer wishes to make reference to a value in store, they simply need to load it into a register (using the "`LDR`" instruction). No DAM Machine mechanism stops them storing a value into this area of memory again (using the "`STR`" instruction). The definitive store only remains definitive if the programmer of an application that includes such a store only changes the store through calls to the "`addtoq`" and "`update`" subroutines of the DAM Machine.

5.3 Data Representation

As already demonstrated in this chapter, one word in computer store can represent either a data value, a pointer to an address in store or a processor instruction. For one data value alone there may be many different interpretations of the pattern of bits, for example: a positive integer, a positive or negative integer with the high bit equivalent to a minus sign, a 32-bit floating point number representation, two positive 16-bit integers, four characters in a string, thirty-two separate Boolean values. The representation of data types in one word is discussed in Section 5.3.1. It may be that 32-bits does not provide a programmer with sufficient level of detail for values that their application requires. For example, values that represent pictures, strings of characters, 64-bit *double* arithmetic and so on. Multi-word data representations are discussed in Section 5.3.2.

5.3.1 One Word Data Representations

The DAM Machine does not *know* through its internal data structures the data types of the values in definitive store. It provides no internal representation for the types of stored data like a lookup table and performs no type checking on whether a sequence of arguments is of appropriate type to pass to an operator subroutine. A programmer has to ensure that all redefinitions are appropriate to their application and type check arguments for their operators prior to calling the "`addtoq`" subroutine of the DAM Machine. This can be considered as a weakness of the DAM Machine as it does not support these features common to a high-level language. However, the intention is that using the DAM Machine is at the same low-level as a programmer developing assembly language code is used to dealing with. The DAM Machine provides an architecture for dependency maintenance that the developer of a high-level language notation can exploit as the object of a compiler

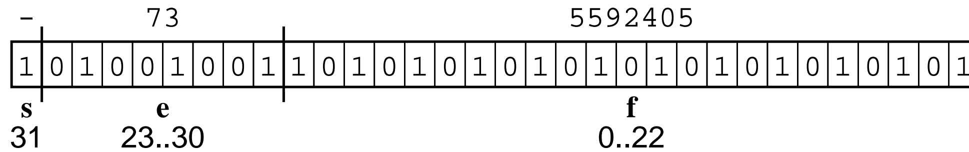


Figure 5.4: IEEE single format representation for a floating point number.

tool.

As long as a programmer implements their operators appropriately and consistently, it is possible maintain dependencies between their chosen data representations if 32-bit words are satisfactory for their application. Figure 5.4 shows a technique for the representation of floating point numbers in 32-bit words. The representation is consistent with the IEEE standard for the representation of single length floating point values [IEE85]. The word of 32-bits is split into 3 bit fields: the bit fraction **f** between $0 \ldots 22$ with 0 the least significant, the biased exponent **e** between $23 \ldots 30$ and the high sign bit **s**. For the range $0 < \mathbf{e} < 255$, the floating point value represented by the bit pattern is $-1.0^{\mathbf{s}} \times 2^{\mathbf{e}-127} \times (1 + \frac{\mathbf{f}}{2^{23}})$. The number represented in the example in the figure is $-1.0^{1} \times 2^{-54} \times (1 + \frac{5592405}{8388608}) \simeq -9.25185 \times 10^{-17}$.

For an application where every word in a definitive store is represented by a single length floating point value and all the operators map between sequences of floating point numbers and a floating point number, this data representation is perfectly adequate. If an application has more than one data representation, such as 32-bit integers combined with floating point, then a programmer needs to keep their own lookup table to determine whether the value at a particular address in store is floating point or integer. They will need subroutines for arithmetic operations on floats and different subroutines for the same operations on integers, for example "*add_float*" for floating point and "*add_int*" for integers.

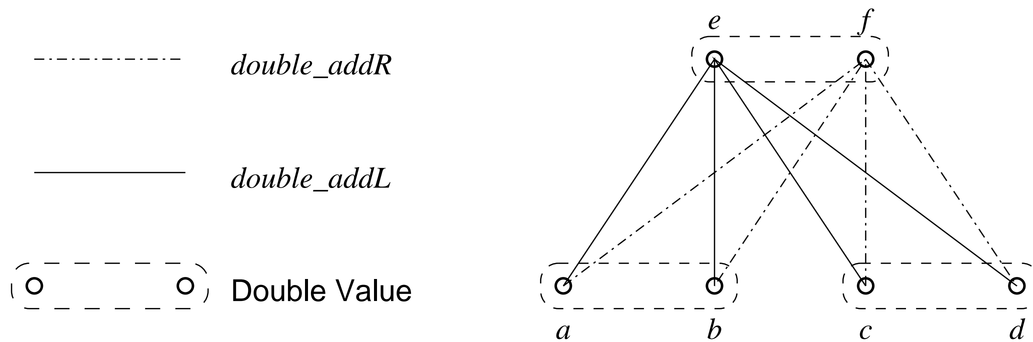


Figure 5.5: Operators for double length floating point operations.

5.3.2 Multi-word Data Representations

What happens when one 32-bit word is not sufficient to store accurately an item of data in an application? Multiple words in definitive store can be joined together to create longer ones in multiples of 32-bits at the discretion of a programmer. The operators represented by blocks of assembly code in the DAM Machine map to a value through the registers of the processor[6]. The registers have a limited length of 32-bits so it is not possible for one operator to maintain dependencies for more than one machine word in definitive store. For a two word (64-bit) data representation, the programmer has the option to implement two specialised operators, one that maps to the highest bits $(32 \ldots 63)$ and one that maps to the lowest bits $(0 \ldots 31)$.

Consider the example shown in Figure 5.5 that represents the addition of two double numbers and maps to another double. Three pairs of nodes in a dependency structure are shown, each pair representing a double length floating point value. The double represented by the pair (e, f) is defined to be the addition of the doubles represented by pairs (a, b) and (c, d). To do this, two operators rather than one are required, one for bits $32 \ldots 63$ of the result called "*double_addL*" and one for bits $0 \ldots 31$ called "*double_addR*". A script of definitions for this dependency structure is shown below.

[6]See Section 5.2.1.

$e = double_addL(a,\ b,\ c,\ d)$
$f = double_addR(a,\ b,\ c,\ d)$

The management of the location of words in definitive store is the task of the programmer of an application. It may be necessary to keep track of both the types of each word and also which words combine to form the value of a definition. For compound data structures such as arrays it becomes necessary to allocate a block in store in which to represent the array data. As operator code is passed a reference to a location in memory, a lookup operator can be defined to map from a base address of an array in definitive store p and an offset q for the qth element of the array, to the value stored at the combined address $p + q$. The code for an operator to do this is shown below[7]:

```
.lookup LDR R3,[R1]     ; Load array offset value into R3 (q)
        MUL R3,R3,#4    ; Calculate word address offset (4q)
        LDR R2,[R0]     ; Load array base value into R2 (p)
        LDR R0,[R2,R3]  ; Load R0 with value stored at p + q
        MOV PC,R14      ; Return control to the DAM Machine
```

In this code, the value stored in definitive store at address p is a pointer to the beginning of an array that is also in definitive store. The dependency structure can maintain dependencies between pointers as a data types as well as between data values. If arrays that change their size are required in an application, the programmer should implement code that is capable of shifting blocks of definitions around, either through reading the definitive store and all its associated data and repeatedly branching to the "`addtoq`" subroutine, or by moving the definitions around themselves ensuring they remain consistent to the internal representations of the DAM Machine.

The representation of data in DAM Machine words requires a programmer to think carefully about the pattern of bits they are to use inside the words to

[7] Note that the addressing of words in RAM is 8-bit "byte by byte". The boundaries of words occur every four bytes, hence the multiply by four line in the code.

hold a data value of a certain type. It does not provide a user friendly interface or parser for a language as the *Tkeden* tool does. Instead, it provides a *Virtual Machine* similar to the Java Virtual Machine [MD97] that can be the target of a compiler's output. With appropriate management of definitive store, the translation process from a high-level language to DAM Machine representation can be used to implement dynamically extendable, open-ended tools for modelling based on scripts of definitions. Unlike the *Java Virtual Machine* and *Tkeden*, the DAM Machine uses only assembled code and replaces the procedural interpretation of *bytecodes*[8] by the maintenance of a block of store consistent with its definition. The move away from interpreted code during the maintenance of dependency leads to more efficient execution of definitive script models.

5.4 The DoNaLD to DAM Translator

The DoNaLD to DAM translator, including a parser for the DoNaLD notation that uses the DAM Machine as its back end dependency maintenance mechanism, was developed by Allderidge as his final year undergraduate degree project [All97], with further detail available in [ABCY98]. The tool developed takes the parser for DoNaLD notations as integrated into the *Tkeden* interpreter and replaces the mechanism for the maintenance of dependency inside the interpreter with the DAM Machine rather than definitions translated into EDEN. The efficiency saving of using the dedicated DAM Machine can achieve up to 20 times better performance in the animation of DoNaLD scripts through redefinition than *Tkeden* running on a similar speed processor.

The process of definition translation from the high-level DoNaLD notation to DAM through an intermediate notation is described in Section 5.4.1. This process

[8]This is done by a huge `case` statement on the Java Virtual Machine and *Tkeden*.

is similar to the stages of the construction of the DM Model from the observations of scripts are presented in Section 4.2.1 of Chapter 4. The degree to which open-ended action is possible with the tool is discussed. Two case-studies for DoNaLD are presented in Section 5.4.2, with a comparison of their performance in animation between the existing *Tkeden* tool and the DAM Machine.

5.4.1 The Translation Process

The DoNaLD to DAM translator is a parser for the DoNaLD notation that operates in two phases. The first phase translates a DoNaLD file into *pseudo-DAM code*, in which each line represents a definition for the DAM Machine. This code is then translated in the second phase by another simple parser into the DAM Machine internal representation by placing appropriate values in the processor registers and then branching to the "`addtoq`" subroutine of the DAM Machine, followed by the "`update`" subroutine. The pseudo code contains some references to special operators that not only represent a dependency but also take an action within the body of their code to alter the graphical image associated with the DoNaLD script, such as a line or a circle.

Figure 5.6 shows an original segment of DoNaLD script for one line in the top window ("`simple`"), its representation is *pseudo-DAM code* in the bottom left hand window ("`scriptfile`") and its graphical representation in the small bottom right hand window ("`simple [default]`"). The figure demonstrates phase one of the translation process. Most lines of the pseudo code contain an operator name for a block of code for an operator in store, followed by an identifier for the definition and a sequence of identifiers for arguments to that operator. Two special operators "`seti`" and "`setf`" are equivalent to the DM Model $value_X$ function. They treat the second argument as the explicit value X for the identifier in the first argument. The other operators shown in the figure are described in the following list. The

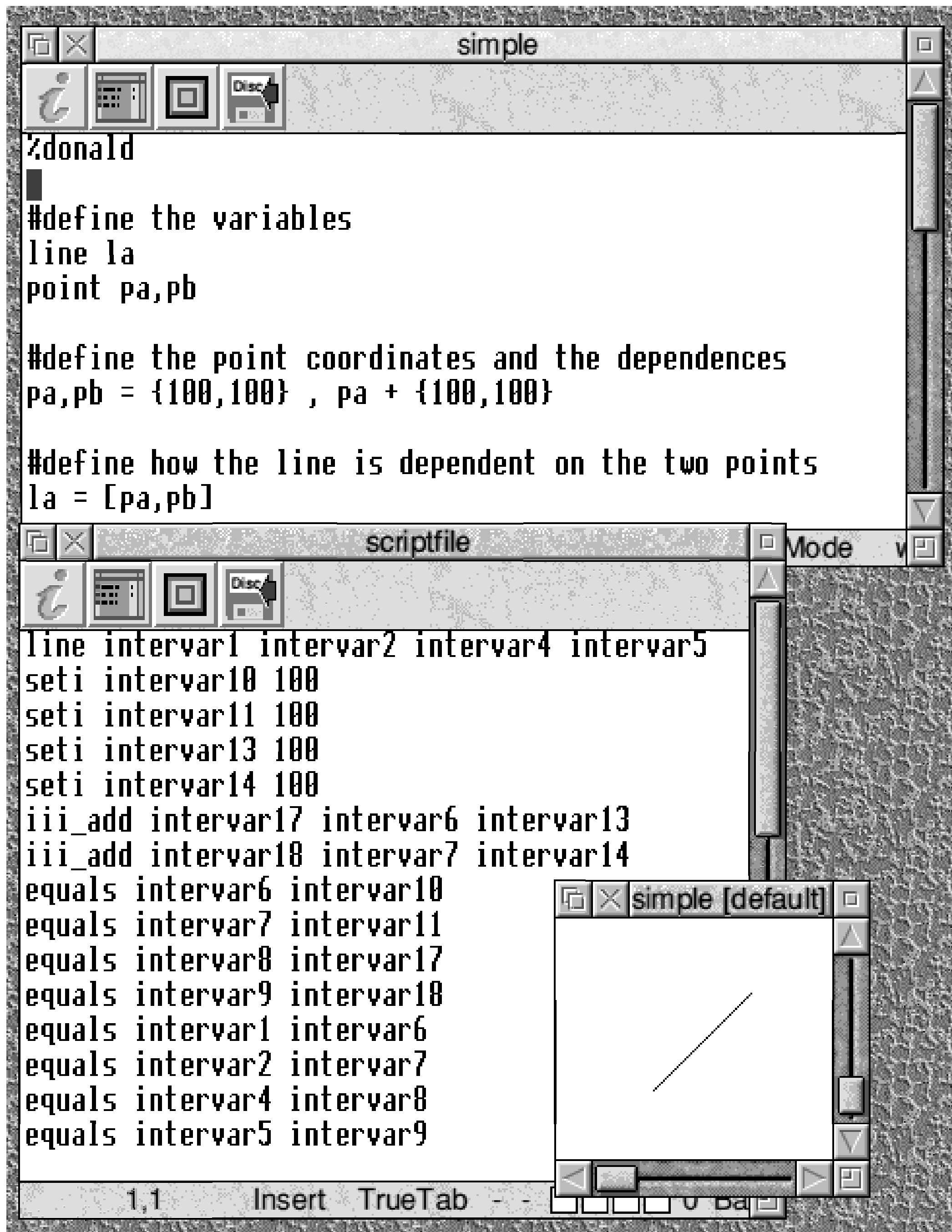


Figure 5.6: Phase 1 of translating a DoNaLD definition for a line into a DAM Machine representation.

strings that start "`intervar`" and end with a number n are identifiers for integer values automatically generated by the parser.

`line` An operator that maps from four integer values to a line defined by two end points. The line is not given an identifier name in the pseudo code and their is no internal one word representation for a *line*. Code inside the block for the operator `line` causes a line to be drawn on the screen.

`equals` A simple operator that defines the value of the identifier second along the line to be equal to the value associated with the identifier third along the line.

`iii_add` This operator adds two integer values together, the values at the locations for the third and fourth identifiers on the line, and places the result in the memory location associated with the second identifier on the line. The "`iii`" stands for an operator that maps from two integers to an integer. Equivalent operators, such as "`fif_add`" for adding an integer and a floating point value and returning an integer, are available to the pseudo code and the DAM Machine implementation of DoNaLD.

The process of translation follows the stages of transformation of a high-level script to make it DM Model representable, with the additional concerns relating to splitting the representation of complex data types such as points into a component representations suitable for store in DAM Machine words. An outline of the stages is given in the enumerated list below, for each definition in a DoNaLD script (a line of the form "*identifier* `=` ...").

1. If the definition contains an expression on its right hand side, build an expression tree. If the expression contains arithmetic for data types that require more than one word to represent them in the DAM Machine, these should be

split into trees for the arithmetic on each component (e.g. split arithmetic for points into two expression trees, one for each dimension).

2. For each parent node of the tree, create an associated reference name (e.g. "`intervar1`"). Each parent node in the tree will become a definition in the pseudo-DAM code. This process removes function composition from definitions and identifies which of the DAM operators can be used to maintain dependencies.

3. If the definition in the DoNaLD is for an explicit value that can be represented by one word in definitive store, create an associated reference name for it. If it is represented by two words in definitive store, create two reference names for it and so on. Store the association between the high-level name and the internal reference name in a lookup table. For implicit definitions, associate the reference root(s) of the related expression tree(s) with the high-level identifier.

4. Write out the pseudo-DAM code, in which every reference name has an associated line in the code and has an appropriate operator for the DAM Machine. The pseudo code contains only internal reference names and no identifiers from the high-level script. Ordering in this script is not important. Explicit values, the leaves of expression trees or sub-components of explicit value definitions in the high-level notation, are defined using the "`seti`" and "`setf`" operators.

Figure 5.7 shows the result of this process for the DoNaLD script shown in Figure 5.6[9]. The dependency structure in the DoNaLD script is represented in the top left hand diagram and the dependency structure in the pseudo-DAM code, and hence the DAM Machine internal representation, is shown on the right hand side. The reference names in the pseudo-DAM that are associated with the identifiers in

[9]A node in the dependency structure in Figure 5.7 labelled *i1* corresponds to the `intervar1` identifier in Figure 5.6.

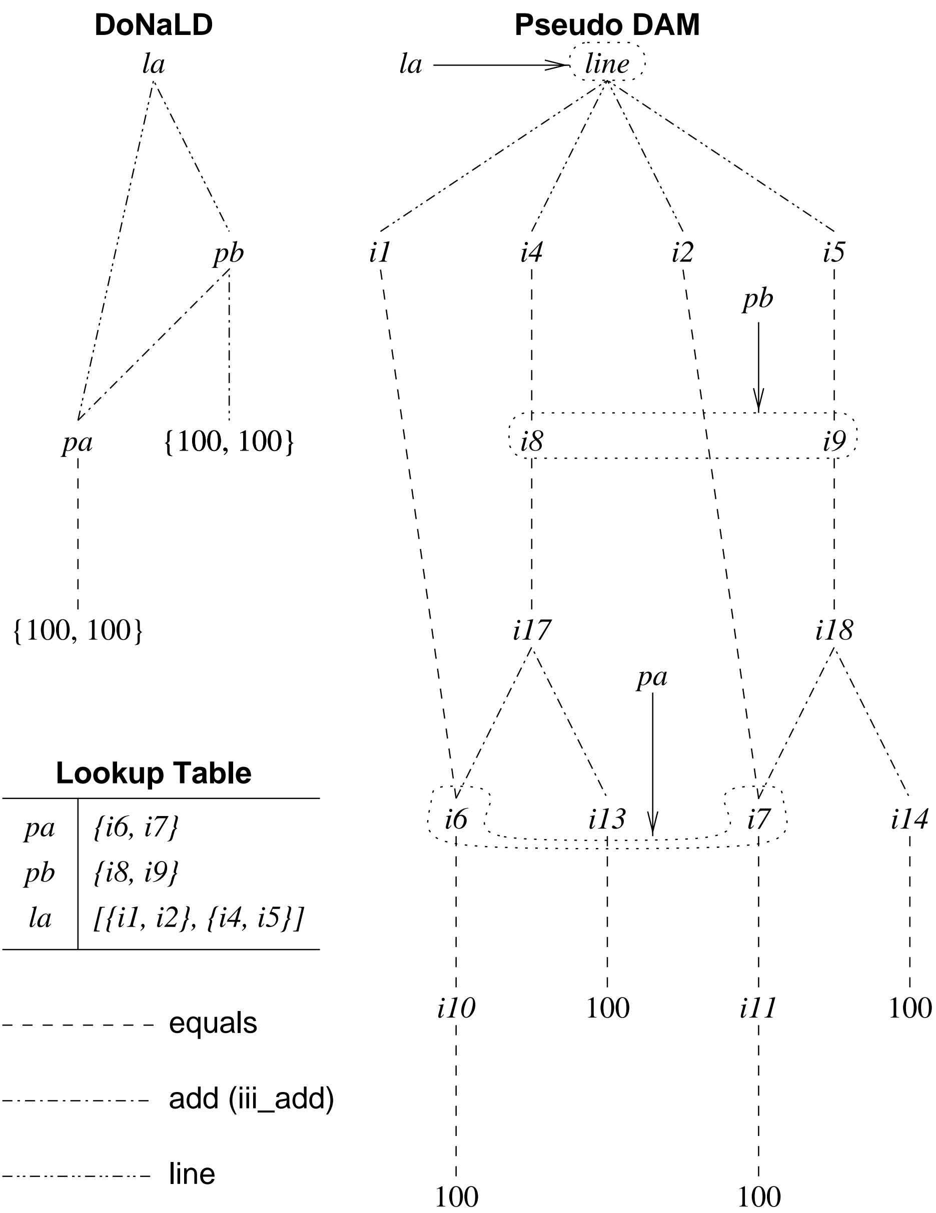


Figure 5.7: Dependency structures for the translation of DoNaLD to pseudo-DAM code.

the DoNaLD script are shown both in the lookup table in the figure and where they are circled by dotted lines in the dependency structure. The steps corresponding to the translation process above for this example script are shown in the following list, with pa and pb considered in order as two separate definitions "$pa = \{100,\ 100\}$" and "$pb = pa + \{100, 100\}$".

Firstly, the process of translation for pa. The expression trees[10] are generated by the existing DoNaLD parser.

1. Identifier pa is declared as a point that is represented by two words in store in the DAM Machine and is therefore split into two expression trees. These are "x = *100*" and "y = *100*".

2. Internal reference names are associated with the parent node of each tree. To do this, x is replaced by *i10* and y is replaced by *i11*.

3. Two reference names (*i6* and *i7*) are created in the translator's lookup table to represent the components of the point pa. The values at the roots of the trees are defined to be equal to the values associated with these new reference names, *i6* equals *i10* and *i7* equals *i11*.

4. The pseudo-DAM code is prepared and is of the form:

```
seti intervar10 100
seti intervar11 100
equals intervar6 intervar10
equals intervar7 intervar11
```

Secondly, the translation process for pb:

[10]Expression trees are represented in the text in the form "*root_node = child_node*" or "*root_node = l_child operator r_child*". The children are either explicit numerical values, existing reference names in the lookup table or sub-trees surrounded by square braces "*[]*".

1. Identifier pb is declared as a point that is represented by two words in store in the DAM Machine and is therefore split into two separate expression trees. As the high-level expression depends on the value of another definition pa, the expression trees reference the lookup table to find the reference names of the component values of pa. The two trees are: "$x1 = i6 + [x2 = 100]$" and "$y1 = i7 + [y2 = 100]$".

2. Internal reference names are associated with the parent node of each tree. To do this, $x1$ and $x2$ are substituted for by $i17$ and $i13$ respectively, similarly $y1$ and $y2$ by $i18$ and $i14$.

3. Two reference names ($i8$ and $i9$) are created in the lookup table to represent the components of the point pb. The values at the roots of the trees are defined to be equal to the values associated with these new reference names, $i8$ equals $i17$ and $i9$ equals $i18$.

4. The pseudo-DAM code is prepared and is of the form:

```
seti intervar13 100
seti intervar14 100
iii_add intervar17 intervar6 intervar13
iii_add intervar18 intervar7 intervar14
equals intervar8 intervar17
equals intervar9 intervar18
```

The result of the production of the pseudo-DAM code is an appropriate set of definitions that represent the same dependencies in the DAM Machine internal store as in the DoNaLD script. In the current implementation, it is possible to re-define explicit integer and floating point values in this structure using the "`addtoq`" and "`update`" subroutines through a graphical user interface, but it is not possible

to make other kinds of new definition or redefinition[11] because the phase 1 translator terminates after reading one file to allow the phase 2 translator to convert the pseudo-DAM code into store. This is technical problem that will need to be researched in the future.

5.4.2 DAM Machine Performance

The DAM Machine is implemented on one platform only and this is a different platform from that on which *Tkeden* is implemented. The performance tests carried out in this section are based on timing the animation of DoNaLD scripts by repeatedly increasing the value of a defining explicit parameter. The DoNaLD code for these scripts is presented in Appendix A.1. The results presented here are not intended as a direct comparison of performance between systems because of their physical differences, but they do demonstrate the potential for fast dependency maintenance on a stand alone computer system running the DAM Machine. When the same DoNaLD script is executed on hardware of similar speed, the DAM Machine can produce a smooth animation where *Tkeden* models appear jerky and slow.

Figure 5.8 shows the DoNaLD graphical output from the script of an engine[12]. The position of the pistons in the script is defined to depend on one variable "`lup`" that determines the current rotation of the engines crank shaft. When an appropriate point is reached in this rotation, the next firing of a spark plug in the firing-order sequence is represented by the temporary appearance of a circle at the top of the cylinder concerned. By repeatedly increasing the `lup` parameter, the location of the pistons inside the engine's cylinders can be animated.

Table 5.1 shows timings for the animation of the engine script and another

[11] Such as all implicit definitions, explicit definitions of points and lines etc..

[12] DoNaLD source for the engine is available is Section A.1 of Appendix A.1. The image is a screen snapshot of the DAM Machine version.

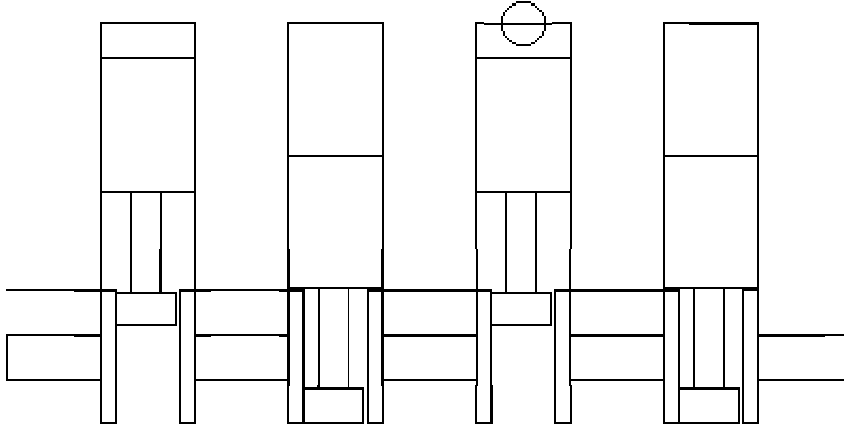


Figure 5.8: Graphical output from the DoNaLD *Engine* script.

Script	ARM710 CPU running RISC OS windows.	ARM710 CPU single tasking.	CY76601 CPU (SPARCstation 2) running X windows.
Engine	26	13	262
Shapes	21	15	172

Table 5.1: Comparative timings for the animation of DoNaLD scripts on an ARM710 processor and a SPARC CY76601 processor.

DoNaLD test script called *shapes*[13]. Timings were carried out for the DAM Machine on an Acorn *Risc PC 700* with an ARM710 processor, clocked at 40MHz and capable of 36 MIPS[14]. A similar specification SUN Microsystems SPARCstation 2 with a CY76601 processor clocked at 40Mhz and capable of 28.5 MIPS was used to run *Tkeden*. Both systems were tested in multi-tasking mode displaying graphical windows interfaces with no other significant tasks running simultaneously.

To show the potential for a single tasking application that uses dependency and is based on the DAM Machine, timings are included for the DAM Machine executing the DoNaLD script with no graphical windows interface running. The application writes directly to the screen memory through low-level system calls. The range and increment value for the parameters repeatedly updated were the same on the Acorn system and the SUN system. Timings shown are an average of three separate timing experiments.

The table shows how the multi-tasking versions of the animation of the engine script produced a factor of 10 improvement for the DAM Machine over *Tkeden* and a factor of 20 to the single tasking version. For the shapes script, which contains more trigonometry and less line drawing, the speed increase is not quite so impressive with a factor of 8 speed increase for the multi-tasking SUN over the Acorn, and a factor of 11.5 for the multi-tasking SUN and the single-tasking Acorn. The visual difference between the two systems is marked. The single-tasking Acorn produces a smoother animation than the slower *Tkeden* model.

[13]DoNaLD source available in Appendix A.1.

[14]The MIPS unit is a measure for the number of Million Instructions Per Second a processor is capable of executing.

5.5 Linking Observation with Visual Metaphor

The concept of linking the state of the internal model with the state of the display of the screen through dependency maintenance was first introduced in the introductory section of this chapter. Figure 5.1 shows how the current state of an experiment to demonstrate Hooke's Law can be represented by a cross positioned on a computer screen. In this section, a case-study is used to show how a block of pixels located in DAM Machine definitive store can be maintained consistent with their individual definitions. They share the same definitive store with other definitions for the model that they represent.

In this case-study, an area of 1600 machine words in definitive store is chosen to represent a 40 by 40 grid of pixels for display on the screen. The value stored at each machine word is 0 if the pixel is black and 1 if the pixel is white. Each pixel is implicitly defined to depend on the value of the function representation[15] of a two dimensional geometric shape, which is sampled at a particular x coordinate and y coordinate. If the pixel is outside the specified shape, as indicated by a negative value of the function representation at the point given by (x, y), it is defined to be black. Otherwise, the pixel is white to represent that it is inside or on the boundary of the shape.

The function representation of two dimensional geometric shapes is achieved in the DAM Machine by special operators that map defining parameters for a shape to a pointer to a subroutine block of machine code. This code evaluates the function representation at a point (x, y) dependent of these parameters. In the case-study, there are three such operators: *circle* for a planar circle shape parametrised by centre and radius, *reactangle* for a rectangle shape parametrised by minimum and maximum x and y coordinates, *cut* parametrised by two pointers to other func-

[15]The function representation for geometric shape is introduced in Section 3.4.2.

tion representations for shape where one shape is cut away from the other (the set theoretical difference operator). The value of (x, y) is passed to the subroutines in registers `R0` and `R1` respectively.

The function representations used in the case-study and implemented as blocks of ARM assembly code for this case-study are specified below.

circle Definition of the form "$circle(r, c1, c2)$". A circle shape centred at point $(c1, c2)$ with radius r. The function representation for the shape f for any point (x, y) is given by the formula

$$f(x, y) = r^2 - (x - c1)^2 - (y - c2)^2 \tag{5.1}$$

rectangle Definition of the form "$rectangle(xmin, xmax, ymin, ymax)$". The rectangle has sides parallel to the x and y axis of the coordinate space, with minimum coordinate along the x-axis of $xmin$ and maximum coordinate $xmax$ etc.. The function representation for the shape f for any point (x, y) is given by the formula

$$f(x, y) = ((x - xmin) \cap (xmax - x)) \cap ((y - ymin) \cap (ymax - y)) \tag{5.2}$$

In this formula, the binary infix operator "$a \cap b$" is used to represent set intersection and in the implementation this is achieved in function representation of shape by finding the minimum value of the two arguments a and b.

cut Definition of the form "$cut(f1, f2)$". The shape represented by a cut definition is the material of the shape with function representation $f1$ with the material of the shape with function representation $f2$ removed. The function representation f for the shape of the cut material at any point (x, y) is given in the formula

$$f(x, y) = f1(x, y) \cap -f2(x, y) \tag{5.3}$$

Each pixel on the screen is defined by the "*evaluate*" operator, which maps a pointer to a function representation *f1* and a sample point (x, y) to 0 or 1. The *evaluate* operator is given by the equation

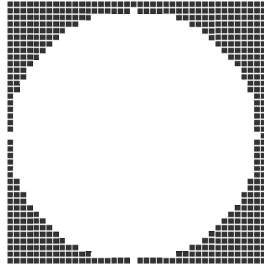
$$evaluate(f1, x, y) = \begin{cases} 0 & \text{if } f1(x,y) < 0 \\ 1 & \text{if } f1(x,y) \geq 0 \end{cases} \tag{5.4}$$

In the case-study shown, the index of a pixel in the array of pixels is used as the point (x, y). The definition of the pixels is set up by code similar[16] to the double *for* loop shown below:

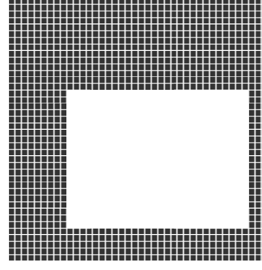
```
for i = 0 to 39 do
   for j = 0 to 39
      do
         "p_ij = evaluate(f1, i, j)"
      done
   done
```

The case-study illustration in Figure 5.9 contains one circular shape, one rectangular shape and has one shape made by the cutting operator. If the function representation for a circle with centre $(19, 19)$ and radius 19 is defined as *f1* with the evaluate operator in the code above, the pixels represent the image labelled "*Circle*" in Figure 5.9. The image is an expanded screen snapshot image of the pixels as displayed by the DAM Machine model. The "*Rectangle*" displayed in the top right hand side of the figure: it has the parametrisation $xmin = 9$, $xmax = 36$, $ymin = 5$ and $ymax = 25$. The rest of the figure concerns a "*Cut*" shape. A section of definitive script that could be used to describe the dependency between the cut shape and the circle and the square is shown below.

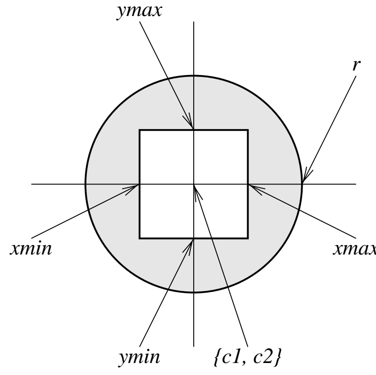
[16] The code used to set up these definitions for the case-study is written in the *BBC Basic* language. The actual code does not illustrate the point of the example as well as the pseudo code presented.



Circle



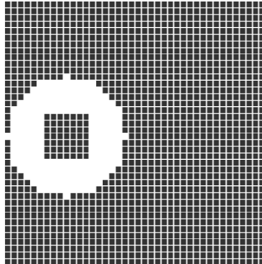
Rectangle



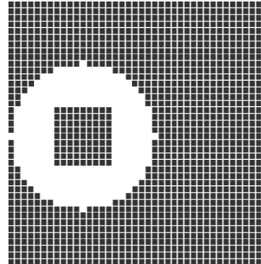
Cut diagram

$$
\begin{aligned}
r &= 9 \\
c1 &= r \\
c2 &= 19 \\
halfr &= half(r) \\
xmin &= subtract(c1, halfr) \\
xmax &= add(c1, halfr) \\
ymin &= subtract(c2, halfr) \\
ymax &= add(c2, halfr)
\end{aligned}
$$

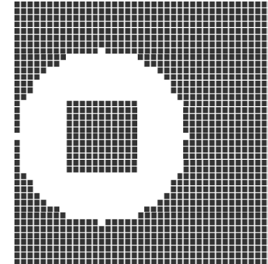
Script for *Cut*



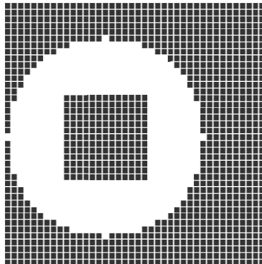
Cut at $r = 9$



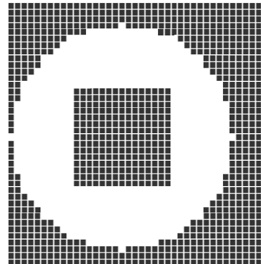
Cut at $r = 11$



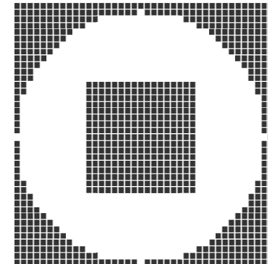
Cut at $r = 13$



Cut at $r = 15$



Cut at $r = 17$



Cut at $r = 19$

Figure 5.9: Direct dependency between screen and script models.

$$
\begin{aligned}
circ &= circle(r, c1, c2) \\
sq &= rectangle(xmin, xmax, ymin, ymax) \\
ct &= cut(circ, sq)
\end{aligned}
$$

This environment allows a user to experiment with models of rectangular shapes cut away from circular shapes. The definitions and diagram shown in Figure 5.9 illustrate some further relationships that can be introduced to the DAM Machine model. The geometry represented by the model is a circle that has a square, with sides the same length as its radius and centred at the centre of the circle, cut away from inside it. The sequence of six screen images shown at the bottom of the figure show the effect of redefining the value of r by calling the "`addtoq`" and "`update`" methods of the DAM Machine. In each case, the image changes through the update propagation through definitive store, not through a procedural sampling of the function representation of the cut. The time for this propagation through 1600 DAM Machine definitions on the 40MHz single tasking ARM710 processor running the DAM Machine is 0.52 seconds for each image.

This case-study demonstrates the concept of creating indivisible relationships between the screen and a model. This creates an open-ended environment in which a modeller can simultaneously construct the model, improve the model, create a visual metaphor for interacting with the model and fine tune the visualisation to their requirements. Visual operations, such as zooming and panning the image, and changes to the parameters to the model are achieved through the same redefinition mechanism. The state of machine words in definitive store remain consistent with their definitions through the DAM Machine and hence the state of the image also remains consistent with its definition. The better the quality of the visual metaphor for the model, the closer the experience of interaction with a model is to actual interaction with the model's referent.