Chapter 4

The Dependency Maintainer Model

4.1 Introduction

Dependency maintenance can be carried out at many levels of abstraction without compromising semantic subtlety of definitive state representation. Data structures can be refined by replacing variables of a higher-level type with families of definitions of lower-level types. State representation is just as effective with lower-level types as with higher-level types, provided that the correct dependency relationships between lower-level data are introduced, and that the pattern of redefinition respects these relationships.

There are many examples of dependency maintenance at different levels. For instance:

- A model of a table can be considered as a point set at a low-level and as a family of features at a higher-level.
- The case-study of modelling a vehicle cruise control system [BBY92] includes

a model for a speed transducer. At a low-level of abstraction, observed dependency for the mechanics of the speed transducer are represented. At some higher-level, the model represents the speed transducer from an agent-oriented perspective.

- A DoNaLD high-level script is translated into an EDEN low-level script. It is possible to corrupt the dependencies expressed by DoNaLD definitions by interacting with them directly in EDEN.

In this chapter, issues relating to the implementation of definitive scripts are studied via one particular strategy of refinement of high-level scripts. This refinement takes a script in an existing high-level notation and transforms it into families of definitions in a low-level definitive script where there is only one data type: integer. These refined scripts can be represented in the framework of the *Dependency Maintainer Model* (DM Model). Mathematical sets and mappings are used to construct the model and to formulate the representation of definitive scripts. Translation to the DM Model establishes a bridge between observables that represent external semantics of a model (including its current value and dependencies) and variables that concern the internal semantics of representing a value on a computer.

A variable can be viewed as equivalent to an observable with no constraint upon its redefinition. The *block redefinition algorithm* can be used to implement protocols for redefinition that are consistent with redefinition of observables in a definitive script. Figure 4.1 shows the relationship between a high level-script of definitions, an integer-only low-level script of definitions with equivalent semantics (S) and a representation of this script by the DM Model ($\mathcal{M}_S$). It is possible to transform a high-level script to the lower-level representation but it is not necessarily possible to construct the high-level script from the low-level script. A DM Model $\mathcal{M}_S$ can represent the low-level script S using sets and mappings in such a way that

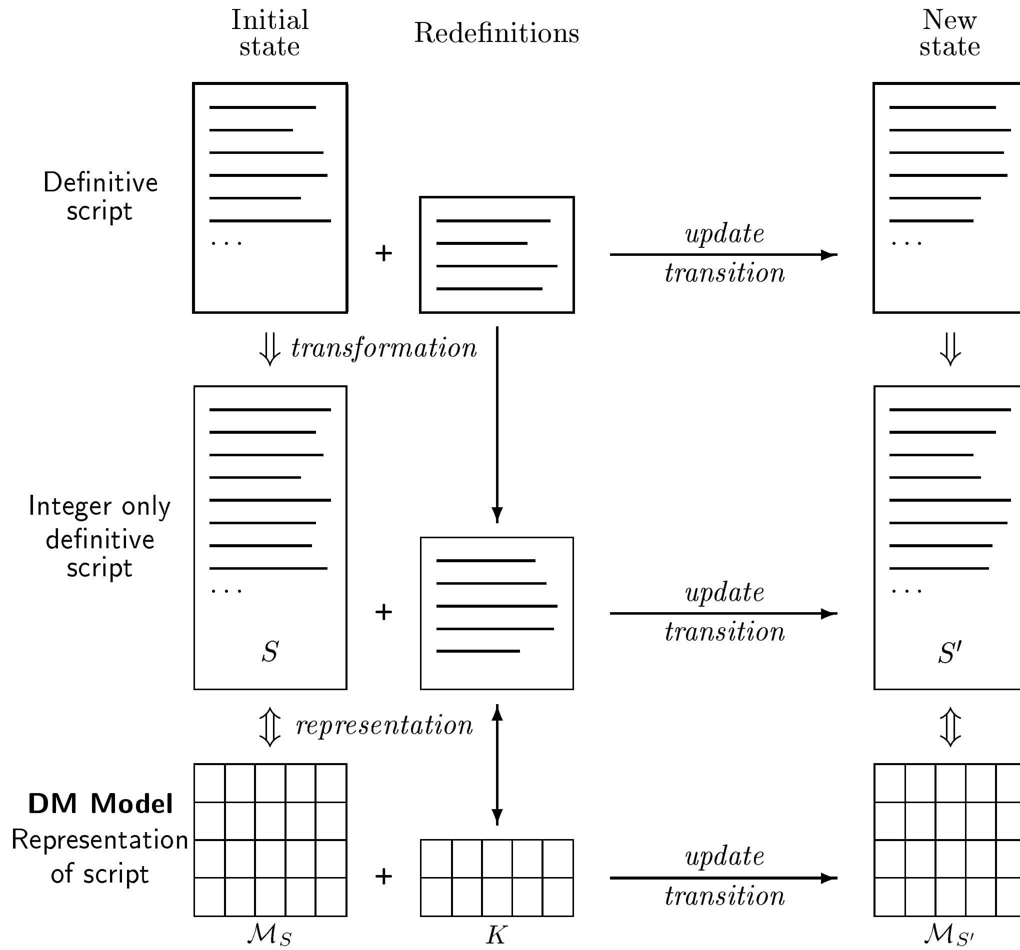


Figure 4.1: Definitive scripts and the DM Model.

the script can be reconstructed from the model. A low-level script such as S is said to be *DM Model representable*.

Redefinition of a high-level script induces an update state transition for the script. The new state of a high-level script can be transformed to the low-level script S', as can the block of redefinitions that resulted in the state transition. As illustrated in Figure 4.1, the low-level block of redefinitions can also be represented by the DM Model. The new state of the low-level script can be represented by a DM Model $\mathcal{M}_{S'}$. The block redefinition algorithm updates the DM Model consistently with the redefinitions to represent the new state. The current value associated with each observable in the low-level script is updated. Updating is the process by which we compute a new value consistent with redefinition.

In this chapter, the transformation process from high-level to low-level script is described. This is followed by an explanation of the representation of low-level scripts as DM Models. This model is used to formalise some important qualities of definitive scripts. These are used as a basis for discussion of strategies for update transition. The block redefinition algorithm is presented with an example of one update state transition. Graphical representations of *dependency structures* are introduced and case-studies use these to illustrate issues relating computational efficiency with dependency maintenance.

4.2 The DM Model for Definitive Scripts

In Chapter 3, the relationship between data structure and dependency is explored. The EDEN notation implements `boolean`, `integer`, `real`, `string` and `list` data types [YY88]. All models written in EDEN must use these types of data to represent their observation specific data. The `list` can often suffice for this purpose although there are some problems with the manipulation of data in lists.

Definitive script	First stage	Value
`a is (b + c) * d`	$a = times(add(b,c),d)$	$a = 48$
`b is power(e, 3)`	$b = power(e,3)$	$b = 8$
`c = 4`	$c = 4$	$c = 4$
`d is 2 * e`	$d = double(e)$	$d = 4$
`e = 2`	$e = 2$	$e = 2$

Table 4.1: A typical definitive script and stage 1 of script transformation.

There are many mechanisms in EDEN to handle functions, triggered actions and dependency maintenance, as described in Section 1.2.1. The motivation for building the DM Model is to strip away the overheads associated with dynamic interpretation of data types and functions, as well as the ordering of triggered procedural actions. Instead, the DM Model concentrates on the fundamental task of dependency maintenance. The only data type considered in the DM Model is the integers $\mathcal{Z}$. In the transformation from high-level scripts to low-level DM Model representable scripts, it is assumed that it is always possible to choose a large enough integer to represent the data types of the high-level script.

4.2.1 Transformation of Scripts

The DM Model is constructed by examining scripts of dependencies and extracting from these some common abstractions that can be used to reason about them. Every definition in a definitive script has an identity (a name) that is associated with both a value and a definition. For the purpose of the construction of a DM Model, it is assumed that the only data type is the integers $\mathcal{Z}$.

Table 4.1 shows an ideal definitive script for illustrating the transformation process, next to a script that represents the first stage of this process. The process of transforming any script to a script that can be represented by the DM Model has four stages. The values that the identities are associated with, consistent with their definition, are shown in the *value* column. From a script of definitions, it is

Reference	Second level of abstraction	Value
α	$a = times(x, d)$	$a = 48$
$\alpha + 1$	$b = power(e, y)$	$b = 8$
$\alpha + 2$	$c = 4$	$c = 4$
$\alpha + 3$	$d = double(e)$	$d = 4$
$\alpha + 4$	$e = 2$	$e = 2$
$\alpha + 5$	$x = add(b, c)$	$x = 12$
$\alpha + 6$	$y = 3$	$y = 3$

Table 4.2: Stage 2 of script transformation.

possible to calculate the value associated with an identifier from its definition and the definition of its dependencies. The DM Model represents both the value and the definition of an identifier simultaneously. The four stages for transforming a script into one that can be represented by the DM Model are listed below.

Stage 1 In a low-level script, every function in the definitions in the high-level script is replaced by a function that is a member of the set $\mathcal{F}$, where

$$\mathcal{F} = \{f \mid f : \mathcal{Z}^* \to \mathcal{Z}\} \tag{4.1}$$

The set $\mathcal{F}$ contains all functions that are mappings from sequences of integers to one integer value. This ensures that all arguments for functions and the resulting domain of these mappings are closed on the set of integers and that the DM Model has only one data type.

The script shown in the example in Table 4.1 requires very little transformation as it already represents dependency between integer values. The choice of how to represent values in the high-level script as integers is unimportant unless it is necessary for it to be possible to reconstruct the high-level script from the integer-only low-level script.

Stage 2 The next stage of transformation is to expand all function composition. In addition, explicit arguments to functions are replaced by adding new identifiers

to the low-level script. An example of stage 2 for the scripts in Table 4.1 is shown in Table 4.2. The definition $b = power(e, 3)$ is replaced by $b = power(e, y)$, where $y = 3$ is a new definition in the low-level script.

Consider the functions g_1 and g_2 that are composed in a script to define the value of observable p to depend on observable q through the definition $p = g_1(g_2(q))$. This example definition is transformed into two definitions $p = g_1(r)$ and $r = g_2(q)$ in a low-level script, where r is chosen as a new identifier in the script.

The way in which an expression such as "`a is (b + c) / d`" is broken down into composed functions and new definitions is not considered in detail here. (For a particular illustration of this process, see Section 5.4.1.) An alternative method for expanding function composition without increasing the number of definitions is to use the function $expr_X$. This function represents the evaluation of a template expression X. For the example definition "`a is (b + c) / d`", the second stage of the transformation of `a` to a definition in a DM Model representable script is

$$a = expr_{(s_1+s_2)/s_3}(b, c, d) \tag{4.2}$$

The definition in equation 4.2 defines the value of a to be equal to $\frac{b+c}{d}$, with the associations to the template expression $s_1 = b$, $s_2 = c$, $s_3 = d$.

Stage 3 The third stage of transformation is to choose a range of integers between α and $\beta \Leftrightarrow 1$ that are to be used as unique integer *reference numbers* for each definition. The size of the range $\beta \Leftrightarrow \alpha$ is equal to the number of definitions after the second stage of transformation of the script. Table 4.2 shows the script from Table 4.1 with a choice of such suitable reference numbers following the expansion of all composed functions. Note that in this example, the values

Transformed Script	Reference $\in [\alpha, \beta \Leftrightarrow 1]$	Function $\in \mathcal{F}$	Dependencies $\in [\alpha, \beta \Leftrightarrow 1]^*$	Value $\in \mathcal{Z}$
$a = times(x, d)$	α	$times$	$(\alpha+5, \alpha+3)$	48
$b = power(e, y)$	$\alpha+1$	$power$	$(\alpha+4, \alpha+6)$	8
$c = value_4()$	$\alpha+2$	$value_4$	$()$	4
$d = double(e)$	$\alpha+3$	$double$	$(\alpha+4)$	4
$e = value_2()$	$\alpha+4$	$value_2$	$()$	2
$x = add(b, c)$	$\alpha+5$	add	$(\alpha+1, \alpha+2)$	12
$y = value_3()$	$\alpha+6$	$value_3$	$()$	3

Table 4.3: Script transformed ready for representation by the DM model.

associated with the identifiers are consistent with the values defined by the high-level script, where the identities are common between the original script and its transformation.

Stage 4 The final stage of transformation to a DM Model representable script is to comply with a DM Model rule that every identifier must have a definition by a function. To achieve this, the generic function $value_i \in \mathcal{F}$ can be used, where for all integer values i and for all sequences of integers S the *value* function maps to i. This can be expressed by the proposition[1]

$$\forall i \in \mathcal{Z}, \forall S \in \mathcal{Z}^* \bullet value_i(S) = i \tag{4.3}$$

A transformed script can be represented by the DM Model. Table 4.3 shows the transformed example script prepared for this representation. The *function* column shows the name of the function used in the definition introduced in stage 1 and 2. The *reference* column shows the reference number in the range $[\alpha, \beta \Leftrightarrow 1]$ associated with each definition, in stage 3 of the transformation. If all the sequences of arguments to the functions that are part of the definitions have their elements

[1]The bullet symbol "•" is used in this chapter to represent *such that* in propositional statement.

substituted by their associated reference numbers, the sequences become as shown in the *dependencies* column.

4.2.2 Representing Definitive Scripts - The DM Model

Every reference number in the range $[\alpha, \beta \Leftrightarrow 1]$ can be mapped to a triple comprising:

- its associated value in $\mathcal{Z}$;
- its defining function in $\mathcal{F}$;
- a sequence of arguments to that function in $\mathcal{A}^*$.

This mapping is illustrated for the running example in the columns of Table 4.3. All identifiers are replaced by their reference number in all parts of the DM Model.

A transformed script S can be represented by a DM Model $\mathcal{M}_S$ that is given by a 4-tuple $(A, F, D, V) = \mathcal{M}_S$. Each element of the tuple is defined as shown below:

A - The range of reference numbers for the script S, where $A = [\alpha, \beta \Leftrightarrow 1]$ and the number of definitions in the DM Model is equal to $\beta \Leftrightarrow \alpha$.

F - A mapping $F : A \rightarrow \mathcal{F}$. Every reference number for a definition in the transformed script is mapped to its associated function by mapping F.

D - A mapping $D : A \rightarrow A^*$. Every reference number for a definition in the transformed script is mapped to its associated sequence of arguments by mapping D. The elements of the sequence are reference numbers, as shown in Table 4.3.

V - A mapping $V : A \rightarrow \mathcal{Z}$. Every reference number for a definition in the transformed script is mapped to the current (integer) value associated with that definition by mapping V.

For the example in Table 4.3, the DM model (A, F, D, V) would be represented by the following sets:

$$
\begin{aligned}
A &= [\alpha, \alpha + 6] \\
F &= \{\alpha \mapsto \mathit{times}, \alpha + 1 \mapsto \mathit{power}, \alpha + 2 \mapsto \mathit{value}_4, \alpha + 3 \mapsto \mathit{double}, \\
&\quad \alpha + 4 \mapsto \mathit{value}_2, \alpha + 5 \mapsto \mathit{add}, \alpha + 6 \mapsto \mathit{value}_3\} \\
D &= \{\alpha \mapsto (\alpha + 5, \alpha + 3), \alpha + 1 \mapsto (\alpha + 4, \alpha + 6), \alpha + 2 \mapsto (), \\
&\quad \alpha + 3 \mapsto (\alpha + 4), \alpha + 4 \mapsto (), \alpha + 5 \mapsto (\alpha + 1, \alpha + 2), \\
&\quad \alpha + 6 \mapsto ()\} \\
V &= \{\alpha \mapsto 48, \alpha + 1 \mapsto 8, \alpha + 2 \mapsto 4, \alpha + 3 \mapsto 4, \alpha + 4 \mapsto 2, \\
&\quad \alpha + 5 \mapsto 12, \alpha + 6 \mapsto 3\}
\end{aligned}
$$

An up-to-date model $\mathcal{M}_S$ is one in which every value is consistent with its definition. To formally define the concept of *up_to_date*, it is first necessary to define the function *lookup*. This maps a sequence of references to a sequence containing the values associated with to these references by DM Model, as given by mapping V. The sequence of values has the same order as the sequence of associated references, where

$$
\mathit{lookup}_V : A^* \rightarrow \mathcal{Z}^*
$$

$$
\mathit{lookup}_V(a_0, \ldots, a_m) = (V(a_0), \ldots, V(a_m))
$$

The boolean condition *up_to_date* — that is true when a script is up-to-date — can now be defined. The value associated with V for every reference number $a \in A$ in DM Model $\mathcal{M}_S$ is given by the evaluation of the associated function

$F(a)$. The arguments to this function are the sequence of *looked up* values for the dependencies of a, with references that are in the sequence $D(a)$. The condition can be expressed as

$$\begin{aligned} & up_to_date(\mathcal{M}_S) \Leftrightarrow \\ & \quad \forall a \in A \bullet V(a) = F(a)(lookup_V(D(a))) \end{aligned}$$

There is a problem determining the up-to-date status of a script if the sequence $D(a)$ contains a, where $D(a) = (a_0, \ldots, a_m) \wedge a \in \{a_0, \ldots, a_m\}$. In this situation, the process of updating $V(a)$ itself invokes an update of $V(a)$. For this to be consistent with the intended semantics of a definitive script, either the definition of $V(a)$ must be deemed invalid or the value of $V(a)$ undefined. In the context of the DM Model, the former convention is adopted.

Note that in the context of a parametric or variational modeller [SM95], a definition is interpreted as a simple form of equational constraint. This means that the equation

$$V(a) = F(a)((V(a_0), \ldots, V(a), \ldots, V(a_m))) \tag{4.4}$$

is acceptable provided that it can be solved for $V(a)$.

A definitive script that contains a reference that directly or indirectly depends on itself is said to have *cyclic dependency*. The next section examines the problem of identifying cyclic dependency in more detail.

4.2.3 Ordering in and the Status of DM Models

Given a DM Model representing a script, it is possible to determine whether the script contains any cyclic dependencies. When there is no cyclic dependency, the references in a DM Model can be partially ordered. This ordering can also be used

to analyse the effect that structures in a model have on the efficiency of updating a script.

For a DM Model $\mathcal{M}_S$, it is possible to formalise the concept of dependency using the relation "$\lhd_D$" on set A. If $x \lhd_D y$ then the value of y is said to be *directly dependent* on the value of x. For two references x and y in A, the relation satisfies the condition

$$x \lhd_D y \Leftrightarrow D(y) = (a_0, \ldots, a_m) \wedge x \in \{a_0, \ldots, a_m\} \tag{4.5}$$

This relation $\lhd_D$ is not necessarily transitive. (A transitive relation $\prec$ over the integers is one for which $\forall x, y, z \in \mathcal{Z} \bullet (x \prec y) \wedge (y \prec z) \Rightarrow (x \prec z)$.) This is shown by the following counterexample. The script $a = 3, b = double(a), c = double(b)$ has DM Model representation with set $D = \{a \mapsto (), b \mapsto (a), c \mapsto (b)\}$. The relations for this model include $a \lhd_D b$ and $b \lhd_D c$. However, there is no relation $a \lhd_D c$ and so there is a script for which "$\lhd_D$" is not transitive.

The relation $\lhd_D$ can be used to define the set of all dependencies of a particular reference a in a model, as well as the set of all dependents for a reference. The direct dependencies of a are the elements of the sequence $D(a)$. The direct dependents of a are the references that have a as a dependency.

For any reference $a \in A$, the set of all references upon which the value of a directly depends — its *direct dependencies* — is given by the function *d_dependencies*, where[2]

$$d_dependencies : A \rightarrow \mathcal{P}(A)$$

$$d_dependencies(a) = \{x \mid x \lhd_D a\}$$

The set of all references that the value a depends on directly or indirectly is given by the function *dependencies*, where

$dependencies : A \rightarrow \mathcal{P}(A)$

[2] $\mathcal{P}(Z)$ represents the power-set (set of all subsets) of the integers $\mathcal{Z}$.

$$dependencies(a) = d_dependencies(a) \cup \left(\bigcup_{x \in d_dependencies(a)} dependencies(x) \right)$$

A very similar construction leads to the formal definition of dependents, references whose associated values should be updated if the value associated with a reference a is updated. The set given by function *d_dependents* contains all the references that have a as a direct dependency, where

$$d_dependents : A \to \mathcal{P}(A)$$
$$d_dependents(a) = \{x \mid a \lhd_D x\}$$

Elements of this set are known as *direct dependents* of a. Members of this set may in turn have additional dependents to the members of *d_dependents* and are *indirect dependents* of a. All the direct and indirect dependents of a are defined as elements of the set given by the function *dependents*, where

$$dependents : A \to \mathcal{P}(A)$$
$$dependents(a) = d_dependents(a) \cup \left(\bigcup_{x \in d_dependents(a)} dependents(x) \right)$$

A DM Model contains a cyclic dependency if there is a reference in the model that is directly or indirectly dependent upon itself. There are two ways to express the boolean condition *cyclic* for a particular DM Model $\mathcal{M}_S$, viz.

$$cylic(\mathcal{M}_S) \Leftrightarrow \exists a \in A \bullet a \in dependencies(a) \tag{4.6}$$

$$cylic(\mathcal{M}_S) \Leftrightarrow \exists a \in A \bullet a \in dependents(a) \tag{4.7}$$

A DM Model is in a *stable* state if all values associated with its references are consistent with their definition (up-to-date) and the model contains no cyclic dependencies. The boolean condition *stable* represents this conjunction, where

$$stable(\mathcal{M}_S) \Leftrightarrow up_to_date(\mathcal{M}_S) \wedge \neg cyclic(\mathcal{M}_S) \tag{4.8}$$

Block of redefinitions	DM Model representation		
	Reference	Function	Dependencies
`a is b * q`	α	*times*	$(\alpha+1, \alpha+7)$
`e = 1`	$\alpha+4$	$value_1$	$()$
`q = c div d`	$\alpha+7$	*div*	$(\alpha+2, \alpha+3)$

Table 4.4: A script of redefinitions and their DM Model representation.

In the context of figure 4.1, it is the stable states of the DM Model that can be interpreted at the higher levels of abstraction. The current status of a DM Model is consistent with the current status of the scripts it represents, both the original script and DM Model representable script. If the DM Model representing a script is *unstable* (not stable), then the script contains at least one cyclic dependency or the values have yet to be successfully updated. To be consistent with high-level interpretation, the state transition effected by an update algorithm should lead to a new stable state for a DM Model.

4.2.4 DM Model State Transitions

The *protected_update* procedure defined in this section performs a single *state transition* (or *update*) of the DM Model, from the current state $\mathcal{M}_S = (A, F, D, V)$ to a new state $\mathcal{M}_{S'} = (A', F', D', V')$. State transitions of the model represent the redefinitions of values and definitions in a transformed definitive script. It is assumed here that any redefinitions that update the DM Model are chosen appropriately to suit the underlying semantics of the transformed and high-level scripts (cf Figure 4.1).

One possible block of redefinitions for the script in Table 4.1 is shown in Table 4.4. Three redefinitions are shown in the left hand column with a DM Model representation of these redefinitions in the right-hand three columns. The process of transforming a block of redefinitions from a general definitive script to a DM Model representable block of redefinitions is the same as the four stage transformation of a script of definitions. If the same identifier exists in the script of definitions and the

block of redefinitions, then it must be chosen to have the same reference number in stage 3.

In a high-level definitive script, the definition associated with an identifier is altered if it exists as a redefinition in a state transition. In a similar way, a state transition from a DM Model $\mathcal{M}_S$ alters the mappings associated with a reference that appears in the block of redefinitions. A block of redefinitions can associate new dependencies with existing references and introduce completely new references into a DM Model. Block redefinition can also make an existing reference depend on a newly introduced reference. For example, after the transition with the redefinition of a in the running example, a dependency on the new identity q with new reference number $\alpha + 7$ is introduced.

For any state of a DM Model, an update can only occur when there is a set of redefinitions K. Each element of this set is a mapping from a reference in a new range of references $A' = [\alpha', \beta' \Leftrightarrow 1]$ to its new definition. The range A' is an extension of the range A. In general, it is possible to extend this range to accommodate any finite number of redefinitions in a block of redefinitions. This extension occurs when references associated with new identifiers for a script of definitions exist in a block of redefinitions. This is illustrated in the example by the introduction of the definition "`q`". It is possible to add to the number of definitions represented in a DM Model through update transitions. It is not possible to remove represented definitions in this way.

A block of redefinitions for a script S with DM Model $\mathcal{M}_S$ and known new range A' known can be represented by a mapping K, where

$$K \subset \{f \mid f : A' \rightarrow \mathcal{F} \times A'^*\} \tag{4.9}$$

In the mapping K, each reference number in the block of redefinitions is mapped to a pair comprising the function part of the redefinition followed by the references for

the sequence of arguments (dependencies) for this function. By way of illustration, the block of redefinitions shown in Table 4.4 is represented by the following mapping K:

$$\begin{aligned} K &= \{\alpha \mapsto (times, (\alpha+1, \alpha+7)), \\ &\quad \alpha+4 \mapsto (value_1, ()) \\ &\quad \alpha+7 \mapsto (div, (\alpha+2, \alpha+3))\} \end{aligned}$$

K_F and K_D will be used to denote the projections of the map K onto the function and domain components of its range respectively.

$$K_F = \{a \mapsto f \mid \exists S \in A'^{*} \bullet (a \mapsto (f, S)) \in K\} \quad (4.10)$$

$$K_D = \{a \mapsto S \mid \exists f \in \mathcal{F} \bullet (a \mapsto (f, S)) \in K\} \quad (4.11)$$

For the purposes of describing the update, it is assumed that the references in K are always well chosen to be in a range $A' = [\alpha', \beta' \Leftrightarrow 1] \supseteq [\alpha, \beta \Leftrightarrow 1] = A$. The set K of redefinitions for a DM Model is *suitable* provided that it contains a new definitions for each additional new reference:

$$\begin{aligned} &suitable(K, A, A') \Leftrightarrow \\ &\quad \forall a \in (A' \setminus A), \exists f \in \mathcal{F}, \exists S \in A'^{*} \bullet (a \mapsto (f, S)) \in K \end{aligned}$$

To define the functions that specify the update process, the "$\oplus$" symbol is used in the same way as in the Z notation [Spi92]. For two relations (mappings) with the same domain and range Q and R, the relation $Q \oplus R$ relates everything in the domain of R to the same objects as R does, and everything else in the domain of Q to the same objects as Q does. The operator dom is introduced to map relations to the sets of objects they reference. For any relation Q, "dom Q" is a set containing all the objects in the domain of Q.

The update of F to F', the reference number to associated defining function mapping of the DM Model, is described by the function upf, where

$$\begin{aligned} \mathit{upf} \quad &: \quad \{g \mid g : \mathcal{Z} \rightarrow \mathcal{F}\} \times \{g \mid g : \mathcal{Z} \rightarrow \mathcal{F}\} \\ &\quad \rightarrow \{g \mid g : \mathcal{Z} \rightarrow \mathcal{F}\} \\ \mathit{upf}(F, K_F) \quad &= \quad F \oplus K_F \end{aligned}$$

Any references common to F and K are removed from F and then these are combined with the new association of functions with references in K. This specification of the update process demonstrates one of the strengths of a DM Model. The set of all possible functions used in the model does not have to be predetermined prior to its initial state.

Similarly, the function upd specifies the state transition from D to D', where

$$\begin{aligned} \mathit{upd} \quad &: \quad \{g \mid g : \mathcal{Z} \rightarrow \mathcal{Z}^*\} \times \{g \mid g : \mathcal{Z} \rightarrow \mathcal{Z}^*\} \\ &\quad \rightarrow \{g \mid g : \mathcal{Z} \rightarrow \mathcal{Z}^*\} \\ \mathit{upd}(D, K_D) \quad &= \quad D \oplus K_D \end{aligned}$$

This function updates the dependencies associated with a reference to be consistent with the redefinitions in set K. Each of the new redefinitions could potentially introduce cyclic dependency into the new state of the model. As an example of this problem, consider the redefinition represented by $K = \{\alpha \mapsto (\mathit{double}, (\alpha))\}$, where the update of a DM Model would result in a reference having a direct dependence on itself.

The new set of values V' associated to references is calculated from the updated functions and dependencies and is not directly related to the set of redefinitions K. After the update transition the DM Model should be up-to-date, where every value is consistent with its definition in the new state. Following the convention introduced in Section 4.2.2, if the new state of D' introduces cyclic dependency

into the model then the update must be deemed invalid. In the absence of cyclic dependency, the *up_to_date* condition should hold for $\mathcal{M}_{S'}$:

$$\forall a \in A' \bullet V'(a) = F'(a)(lookup_{V'}(D'(a))) \tag{4.12}$$

To achieve this, it is necessary to update the values all references associated with redefinitions and their dependents. For a given block of redefinitions K, the set of references whose value is updated is $U \equiv U(K)$, where

$$U(K) = \text{dom } K \cup \left(\bigcup_{x \in \text{dom } K} dependents(x) \right) \tag{4.13}$$

A protected update is one where for any given block of redefinitions K, the new state of the model following an update is still a stable state for a DM Model, even if this means that the block of redefinitions is ignored. To achieve this, the *protected_update* mapping from a DM Model and a block of redefinitions to a new state of the DM Model either has a suitable set of redefinitions that take it to a stable state or the redefinitions are rejected and the state transition does not affect the DM Model. This protection is formalised by the mapping *protected_update*, where $F' = upf(F, K_F)$, $D' = upd(D, K_D)$ and

$$protected_update(\mathcal{M}_S, K) = \begin{cases} \mathcal{M}_S & \text{if } cyclic((A', F', D', V)) \\ & \text{or } \neg suitable(K) \\ (A', F', D', V') & \text{otherwise} \end{cases} \tag{4.14}$$

The values of all the references in U should be updated once K is shown to be suitable and the potential introduction of cyclic dependency by the block of redefinitions has been ruled out. The most efficient ordering in which to update the values associated with the references in U for the condition $up_to_date(\mathcal{M}_{S'})$ to be true is discussed in Section 4.3.2. For a block of redefinitions K, the update of a DM Model $\mathcal{M}_S$ to its new state $\mathcal{M}_{S'}$, which is consistent with the update of the

Script Represented	Reference $\in [\alpha', \beta' \Leftrightarrow 1] = A'$	Function $\in \mathcal{F}$	Dependencies $\in A'^*$	Value $\in \mathcal{Z}$
$a = times(d, q)$	α	$times$	$(\alpha+1, \alpha+7)$	4
$b = power(e, y)$	$\alpha+1$	$power$	$(\alpha+4, \alpha+6)$	1
$c = value_4()$	$\alpha+2$	$value_4$	$()$	4
$d = double(e)$	$\alpha+3$	$double$	$(\alpha+4)$	2
$e = value_1()$	$\alpha+4$	$value_1$	$()$	1
$x = add(b, c)$	$\alpha+5$	add	$(\alpha+1, \alpha+2)$	5
$y = value_3()$	$\alpha+6$	$value_3$	$()$	3
$q = div(c, d)$	$\alpha+7$	div	$(\alpha+2, \alpha+3)$	2

Table 4.5: Example of an updated DM Model.

definitive script S that it represents to a new state S', is

$$\mathcal{M}_{S'} = protected_update(\mathcal{M}_S, K) \tag{4.15}$$

An example of the state of the DM Model after an update is shown in Table 4.5. This example is based on the running example through this chapter (the transformed script in Table 4.3 and the block of redefinitions in Table 4.4). The definition of a has been altered, breaking its dependency to x (reference number $\alpha+5$) and adding a dependency to the new definition q (reference number $\alpha+7$). The definition of e has also changed so that its value is defined by a different *value* function. A new definition has been added so that the range A' is now $[\alpha', \beta' \Leftrightarrow 1] = [\alpha, \alpha+7]$. Six out of the nine values are updated as a result of the state transition, and $\mathcal{M}_{S'}$ is a stable model.

4.2.5 Incremental Construction of DM Models

The DM Model is constructed by the transformation and representation of scripts that already exist. It is possible to construct a DM Model from scratch in the same way that a definitive script can be incrementally constructed from scratch. In this case, the initial state for any DM Model is represented by $\mathcal{M}_\emptyset = (\emptyset, \emptyset, \emptyset, \emptyset)$, a 4-tuple containing four empty sets $\emptyset$. Movement to the next state $\mathcal{M}_S$ is achieved in the

same way as any state transition for the DM Model, through the introduction of a block of redefinitions K that is a representation of the first script S for the new DM Model. This initialization is possible by application of an algorithm such as the block redefinition algorithm presented in Section 4.3.3.

There are no preconceptions about the script, the structure of the script, or its DM Model representation prior to the incremental construction of a model. The initial reference range, functions, sequences of dependencies and values do not have to be predetermined before the initial script of (re)definitions K is known. The process of initialising a DM Model is the same as the process of updating it through state transition. Note that this is consistent with the operation of the existing *Tkeden* interpreter.

4.2.6 Interaction Machines

Wegner introduces the concept of *interaction machines* in [Weg97] and claims that these are more powerful than rule-based algorithms. These are defined as Turing machines extended by the addition of input and output actions that support dynamic interaction with an external environment. The DM Model can be viewed as an interaction machine, with input in the form of redefinitions and output in the form of current values and script represented after update state transitions. A script can be changed beyond recognition by a suitable block of redefinitions. A rule-based algorithm running on a Turing machine cannot be modified until its execution has terminated.

Wegner argues for the radical notion that interactive systems are more powerful problem-solving engines than algorithms and proposes a new paradigm for computing based around the unifying concept of interaction. He notes that interfaces to open systems often constrain interactive behaviour and restrict constituent interface components to goal-directed behaviour. This is because the construction

of these interfaces is necessarily algorithmic using current programming paradigms. Wegner's vision is closely connected with the conflation of computer programming and computer use discussed in Chapter 1 1.1. This conflation of roles has been reasonably successfully demonstrated in the use of the EDEN interpreter (cf the case-study in Section 2.3.2). The work in this thesis is principally concerned with yet another conflation of roles whereby the user and the programmer can develop the computer as an instrument. This conflation is beyond the scope of EDEN.

4.3 Algorithms for DM Machine Update

The DM Model is defined by mappings over sets that represent a script of definitions. This section examines algorithms that both effect a state transition for a script and ensure that the values in the DM Model are up-to-date following the state transition. The new *block redefinition algorithm* for performing the *protected_update* function and transition from V to V' for any DM Model is presented.

The concept of a *dependency structure* is for a DM Model introduced and a graphical representation of these structures is described. This structure is used to explain informally the benefits of the block redefinition algorithm in comparison to previous strategies for the maintenance of dependencies. Dependency structure is also used to illustrate one state transition using this algorithm.

4.3.1 Dependency Structure

The relation "$\lhd_D$" on the set A for DM Model $\mathcal{M}_S$ defines a *dependency structure*. Dependency structures can be used to analyse patterns of dependency in represented scripts. It is possible to draw graphical representations of dependency structures that can be used as cognitive artefacts for the exploration of dependencies in a script. This can inform the choice of redefinitions or the construction of new definitive

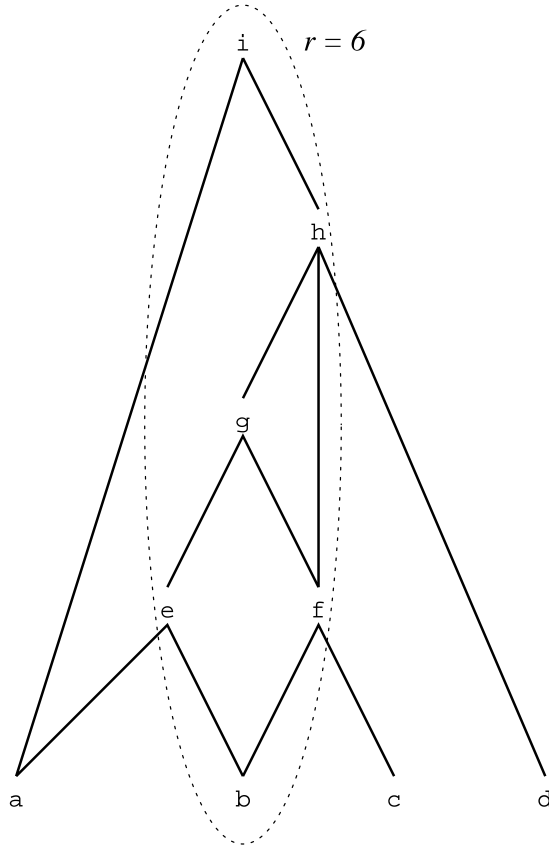
scripts to represent similar observed dependencies in the real world.

The script S represented in Figure 4.2 is used as a new running example to demonstrate the drawing of dependency structures and the application of the block redefinition algorithm. In this figure, the combinatorial graph depicts the *dependency structure* over references in a DM Model $\mathcal{M}_S$ of S. The relation $\lhd_D$ for $\mathcal{M}_S$ is represented on the diagram by edges between the vertices of the graph. An edge exists between vertices with associated reference numbers x and y in the graph if $x \lhd_D y$. If this is the case, vertex y is the parent of child node x. In this representation, the graph is directed and acyclic with directional arrows replaced by orientation up the page. The table in the figure depicts the example script S together with the dependency relations on the set A.

4.3.2 Comparing Update Strategies

In this section, the strategy for ordering the update of the values associated with references in the DM Model (the set V) is considered. The arguments presented are based on the premise that each update is expensive in comparison with the calculation of the update ordering. It could be the case that every data item mapped to by V has been chosen in transformation to be a large integer that represents a complex piece of geometry in a high-level definitive script for geometry. The computation of the functions in F may be computationally expensive for these representations. For example, if an integer represents arbitrary point sets, then an operator such as a blend of two such point sets and the rendering action that produces an image of the blend is likely to be a processor intensive task for a computer system. In such a context, there is a string motivation to determine the optimal number of updates required to effect a particular state transition.

The DM Model representing a transformed script is in a stable state $\mathcal{M}_S$ if all values of words in definitive store are consistent with their definition. During the



Script S	The relation $\lhd_D$ on S	
$a = 3$	$a \lhd_D e$	$a \lhd_D i$
$b = 10$	$b \lhd_D e$	$b \lhd_D f$
$c = 7$	$c \lhd_D f$	
$d = 9$	$d \lhd_D h$	
$e = add(a,b)$	$e \lhd_D g$	
$f = add(b,c)$	$f \lhd_D g$	$f \lhd_D h$
$g = times(e,f)$	$g \lhd_D h$	
$h = max_3(g,f,d)$	$h \lhd_D i$	
$i = power(h,a)$		

Figure 4.2: A Dependency Structure for S

state transition to state $\mathcal{M}_{S'}$ the values of all references contained in the domain of mapping K, considered as *redefinitions*, must be updated. All values associated with references in the set $U(K)$ corresponding to K must also be updated[3], as these values may have changed. One possible strategy to perform these calculations is to update the value of each reference repeatedly until it is observed that condition $up_to_date(\mathcal{M}_{S'})$ holds. This is inefficient as if there are $n = \| A' \|$ vertices in a dependency structure then this method requires $\mathcal{O}(n^2)$ updates to guarantee the DM Model reaches a stable state. For this update strategy, each value has to be calculated n times to guarantee consistency in the worst case.

Knuth [Knu68] presents an algorithm in for topological sorting of the elements of a partially ordered set. In the context of the DM Model $\mathcal{M}_S$, Knuth's algorithm determines an efficient ordering for updating the value of every reference in A'. This reduces the number of updates required to n, as, once the sort has taken place, each value only has to be updates once. However, it is often the case that the block of redefinitions K is smaller than the reference range A'. In this case, it may not be necessary to update all n values, as $\| U(K) \| \leq n$.

In the use of the existing *Tkeden* interpreter, the size of the sets of redefinition is typically of the order of one or two. The interpreter itself can only handle one redefinition at a time ($\| K \| = 1$) and is optimised for this. It often performs a large number of unnecessary calculations. Each Tkeden single redefinition initiates a state transition, even though several redefinitions (a block) presented to the interpreter at the same time may conceivably pertain to the same change of state in the observed model. A better strategy is to consider one state transition as defined by a block of one or more simultaneous redefinitions. In the context of Figure 4.1, a single redefinition in a high-level script may be transformed into a block of redefinitions at the low-level. This fact gives strong motivation for the block redefinition algorithm.

[3] Consider the propagation of change in a spreadsheet after changing the value in one cell.

The useful measure $r \equiv r_S$ can be associated with the dependency structure of a script S. The purpose of r_S is to describe an upper bound on the number of updates associated with a single redefinition in S. The measure r is formally defined as

$$r_S = \max_{a \in A}\{(\parallel \mathit{dependents}(a) \parallel +1)\} \tag{4.16}$$

As a reference in a script can have at most $n \Leftrightarrow 1$ dependents, the value of r_S is bounded by and often significantly less than n. For the script in Figure 4.2, $r_S = 6$.

In *Tkeden*, the absence of block redefinition means that the redefinitions in K are processed one-by-one. This process results in $\parallel K \parallel$ state transitions. The number of associated recalculations of values in U for K has an upper bound of $r \parallel K \parallel$. For a dependency structure with a large value for r, it is possible for this upper bound to exceed the n updates required to update all the values of A'.

To determine an ordering for efficient update where only small subsets of the model are redefined requires the modification of an algorithm such as Knuth's topological sort. The algorithm for efficient update of the DM Model is called the *block redefinition* algorithm, where the word *block* signifies that the algorithm considers several redefinitions simultaneously. This algorithm finds a sensible ordering of work to be done, in the context of both the current dependency structure for the script represented all the redefinitions in the block of redefinitions. The upper bound for the number of updates is at worst n and the actual number of updates is often lower, depending on the context of the current dependency structure and the block of redefinitions.

By way of illustration, consider the example shown in Figure 4.2. A sensible order for updating the values of all references would follow the topological sort a, b, c, d, e, f, g, h and i. If the two redefinitions

$$d \quad = \quad 23$$

$$h = min(g, c, d)$$

are redefined in one state transition, it is only necessary to update (set the value of) d, update the new value of h and then recalculate i. The values of a, b, c, e, f and g all remain the same.

4.3.3 The Block Redefinition Algorithm

The block redefinition algorithm, developed by adapting the Knuth algorithm for topological sorting, has three distinct phases. For DM Model $\mathcal{M}_S = (A, F, D, V)$:

- the first phase updates D and F to D' and F' respectively, so that they are consistent with the redefinitions in K. There is no check for cyclic dependency in this phase.
- the second phase calculates the optimal order in which the values of V need to be updated so that the condition $up_to_date(\mathcal{M}_{S'})$ is true. During this phase, cyclic dependencies can be detected. If cyclicity is detected, the changes made in phase one are revoked and $\mathcal{M}_{S'}$ is set to be the same as $\mathcal{M}_S$, as for the *protected_update* function.
- the third phase performs the updates (step 6 of phase 3) in the order determined in the second phase.

In the specification of the algorithm, an additional mapping $kc : A' \rightarrow \mathcal{Z}$, complementary to those that define the DM Model, is used to introduce numerical counters for each reference. These counters are referred to as the *Knuth counters* as they resemble the counters used in Knuth's topological sort algorithm. In the exposition of the algorithm, it is assumed that, with an appropriate choice of data structures to represent the DM Model mappings, the calculation of the function

d_dependents is trivial. A suitable data representation for this purpose is presented in the description of the DAM Machine implementation in Chapter 5.

The following definitions and notations are used in presenting the block redefinition algorithm:

- a function *seq* to construct an enumeration of a sequence. The sequence $seq(Q)$ is $(q_1, \ldots, q_{||Q||})$ and every element of the sequence is a unique member of the set Q.

- the standard list processing functions *head*, *tail*, *append*, *length* and *contains* on sequences. These functions are: *head* maps to the first element of a sequence, *tail* maps to a sequence with the head element removed, *append* adds a new element onto the end of a sequence, *contains* is a boolean function to test whether a sequence contains a particular element.

$$head((t_1, \ldots, t_n)) = t_1 \tag{4.17}$$

$$tail((t_1, \ldots, t_n)) = (t_2, \ldots, t_n) \tag{4.18}$$

$$append((t_1, \ldots, t_n), x) = (t_1, \ldots, t_n, x) \tag{4.19}$$

$$length((t_1, \ldots, t_n)) = n \tag{4.20}$$

$$contains((t_1, \ldots, t_n), x) \Leftrightarrow \exists k \in [1, n] \bullet x = t_k \tag{4.21}$$

The three phases of the block redefinition algorithm are described step-by-step below. The state of the model prior to execution is $\mathcal{M}_S = (A, F, D, V)$ and the block of redefinitions is K. The new range A' is known prior to algorithm execution. The new state of the DM Model after the algorithm execution for one state transition is $\mathcal{M}_{S'} = (A', F', D', V')$.

Phase 1 1. $kc := \{a \mapsto b \mid a \in A', b = 0\}$.

2. $B := seq(\text{dom } K)$.

3. for $j = 1$ to $length(B)$

   do

   $F := F \oplus \{B_j \mapsto K_F(B_j)\}$.

   $D := D \oplus \{B_j \mapsto K_D(B_j)\}$.

   done

4. Continue to step 1 of phase 2.

Phase 2 1. $a := head(B)$.

2. $C := seq(d_dependents(a))$.

3. for $j = 1$ to $length(C)$

   do

   $kc := kc \oplus \{C_j \mapsto max(kc(a) + 1, kc(C_j))\}$

   if $\neg contains(B, C_j)$

   then $B := append(B, C_j)$

   done

4. $B := tail(B)$.

5. if $kc(a) \geq (\beta' \Leftrightarrow \alpha')$

   then report a *cyclic dependency*, set $\mathcal{M}_{S'} = \mathcal{M}_S$ and halt.

6. if $length(B) > 0$

   then return to step 1 of phase 2,

   else continue to step 1 of phase 3.

Phase 3 1. $B := seq(\text{dom } K)$.

2. for $j = 1$ to $length(B)$

   do (β' is used as a marker.)

   if $kc(B_j) \neq 0$ then $B_j := \beta'$.

done

3. $a := head(B)$.

4. If $a := \beta'$ then skip to step 8 of this phase.

5. $C := seq(d_dependents(a))$.

6. $V := V \oplus \{a \mapsto F(a)(lookup_V(D(a)))\}$.

7. for $j = 1$ to $length(C)$

   do

   if $(kc(C_j) = (kc(a) + 1)) \wedge (\neg contains(B, C_j))$

   then $B := append(B, C_j)$

   done

8. $B := tail(B)$.

9. if $length(B) > 0$

   then go to step 3 of phase 3,

   else go to the end.

End $F' = F$, $D' = D$, $V' = V$ and $M_{S'} = (A', F', D', V')$.

The detection of cyclic dependency in phase 2 is simplistic[4]. At the end of phase 2, the values of the Knuth counters for each reference indicate the order in which the values should be updated, starting at 0. References that have the same Knuth counter after phase 2 can have their values safely updated in parallel.

If there is a cyclic dependency in the model, then the value of a Knuth Counter associated with a reference will eventually be incremented to a value greater than the total numbers of vertices in the dependency structure. This way in which this process operates in illustrated in Figure 4.3, where the Knuth counters in the

[4] i.e. it uses an indirect and relatively inefficient method to identify a simple cyclic dependency such as a self-reference.

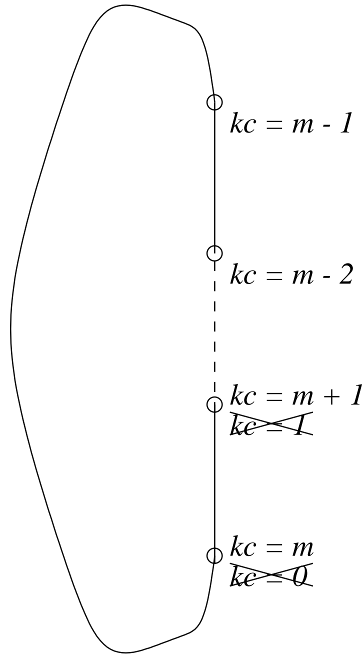


Figure 4.3: The numbering of a structure containing cyclic dependency.

loop of nodes increment repeatedly. In this case, the Knuth counter calculation in phase 2 of the block redefinition will terminate the algorithm because of the assignment of the value $\beta \Leftrightarrow \alpha$ to a counter.

4.3.4 Example of the Block Redefinition Algorithm

Figures 4.4, 4.5, 4.6 and Table 4.6 trace a sample execution of the block redefinition algorithm for one state transition that updates script S shown in Figure 4.2. The model corresponding to this structure, $\mathcal{M}_S = (A, F, D, V)$, is represented in Figure 4.4a, where the identifiers associated with the vertices in the dependency structure are shown in the left-hand-side of each split box representation and the current value of the Knuth counter associated with the identifier is shown on the right-hand-side. In this example, the sets that make up the DM Model 4-tuple

representing the script are as follows[5]:

$$
\begin{aligned}
A &= [a, i] \\
F &= \{a \mapsto value_3, b \mapsto value_{10}, c \mapsto value_7, d \mapsto value_9, \\
&\quad e \mapsto add, f \mapsto add, g \mapsto times, h \mapsto max_3, i \mapsto power\} \\
D &= \{a \mapsto (), b \mapsto (), c \mapsto (), d \mapsto (), e \mapsto (a, b), \\
&\quad f \mapsto (b, c), g \mapsto (e, f), h \mapsto (g, f, d), i \mapsto (h, a)\} \\
V &= \{a \mapsto 3, b \mapsto 10, c \mapsto 7, d \mapsto 9, e \mapsto 13, f \mapsto 17, \\
&\quad g \mapsto 221, h \mapsto 221, i \mapsto 10793861\}
\end{aligned}
$$

The block of redefinitions to be introduced is

$$K = \{a \mapsto (value_2, ()), d \mapsto (double, (c)), g \mapsto (add, (e, f))\} \tag{4.22}$$

The execution of the algorithm will perform the transition of the DM Model from state $\mathcal{M}_S$ to $\mathcal{M}_{S'}$.

Figure 4.4 is used to explain phase 1 of the block redefinition algorithm and is split into three separate sections labeled **a**, **b** and **c**.

a - a graphical representation of the dependency structure of $\mathcal{M}_S$, with all Knuth counters initialised to zero.

b - At step 2 of phase 1, $B = seq(\text{dom } K)$ is the sequence (a, d, g). Vertices in the structure that are in the initial sequence B are highlighted with boxes that have thicker borders.

c - By the end of phase 1, D and F are updated. The redefinition of d in K has changed the topology of the dependency structure so that now $c \lhd_D d$. The values in V are no longer consistent with their definition $(\neg up_to_date(\mathcal{M}_S))$.

[5]Function definitions are as follows: $add(x, y) = x + y$, $times(x, y) = xy$, $max_3(x, y, z) = max(max(x, y), z)$, $power(x, y) = x^y$ and $double(x) = 2x$. For clarity of the exposition, identifiers are used in place of their associated reference numbers.

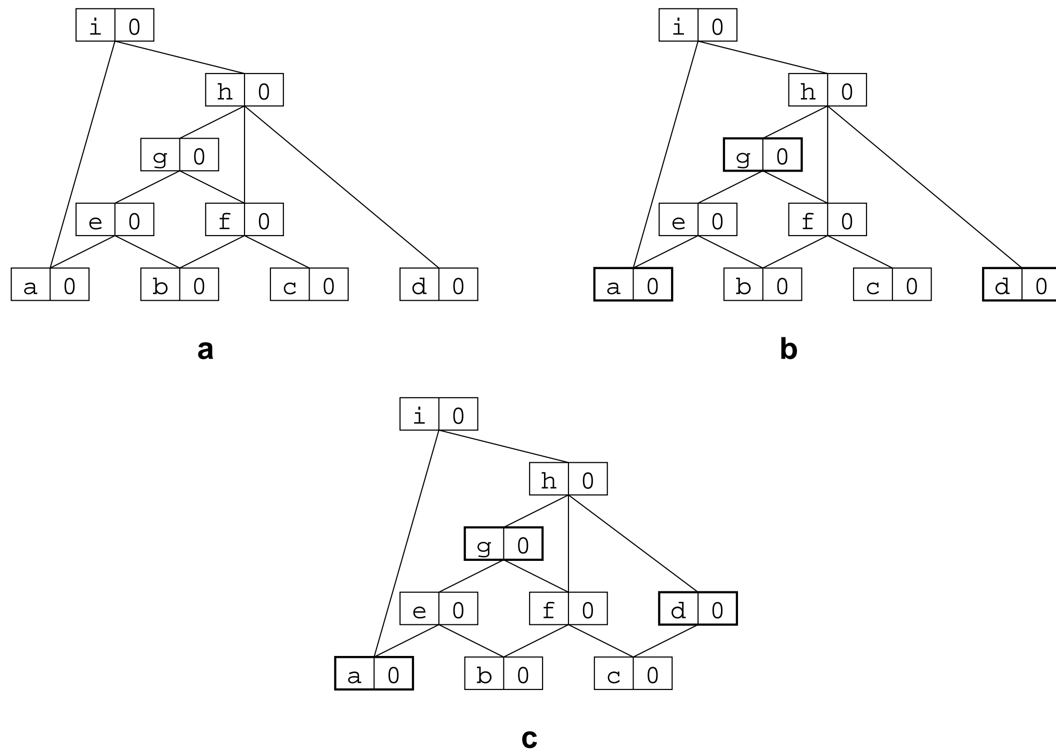
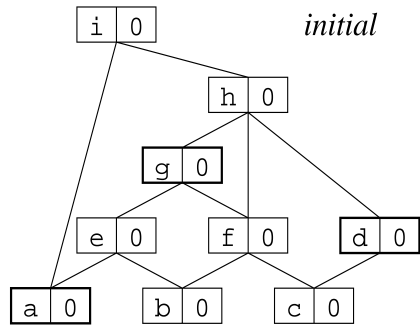
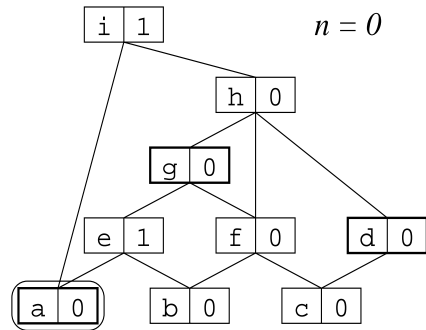


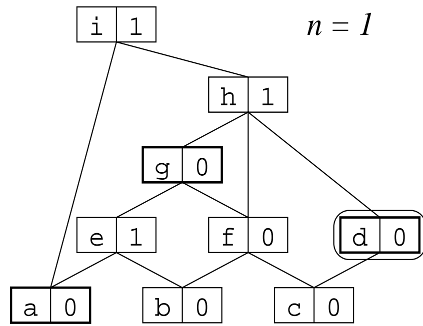
Figure 4.4: Phase 1 of the block redefinition algorithm.

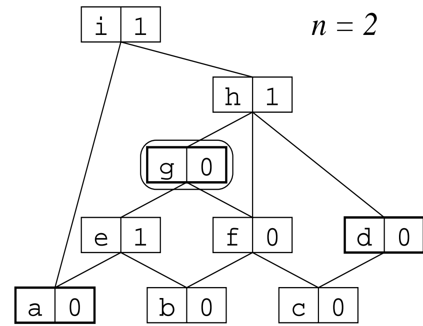
It is possible that a cyclic dependency has been introduced into the structure during phase 1. This is detected in phase 2, which is represented in nine dependency structures across Figures 4.5 and 4.6. In the example, the set of redefinitions K does not introduce any cyclic dependency. The *initial* state for phase 2 is depicted in the top left-hand graph of Figure 4.5. Each graph in the figures is annotated by a value of n, which indicates that it represents the nth iteration of phase 2 of the block redefinition algorithm. For each iteration, the graph representation is a snapshot after step 5 of the phase. The state of current values of sequences B and C are shown below each dependency structure. The current value for a in the algorithm, the head of the sequence B at the start of an iteration of the phase, is depicted by a circled node.

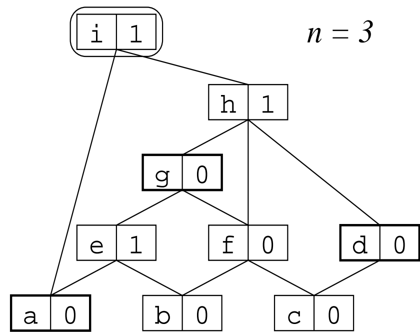
$n = 0$ - Vertex $head(B) = a$ is the first element in the sequence B. C is a sequence











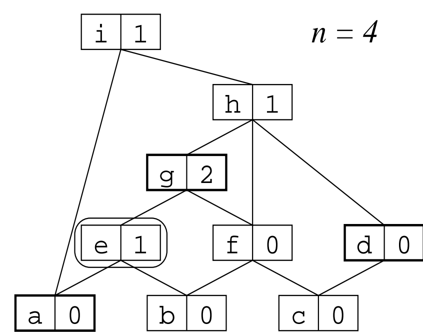


Figure 4.5: Intermediate states after n-th iteration in phase 2 of the block redefinition algorithm ($0 \leq n \leq 4$).

of the dependents of a, which in the example are i and e. Each of these have a Knuth Counter of 0 that is incremented in this first iteration of the phase to $kc(a) + 1 = 1$. As B does not yet contain i or e, these are appended to the end of the sequence. The head element is removed from sequence B before the next iteration.

$n = 1$ - Similar procedure to $n = 0$, except that $head(B) = d$ and $C = (h)$. The only direct dependent of d is node h that has its Knuth counter set to 1. As the sequence B does not contain h, it is appended to the end of B.

$n = 2$ - $head(B) = g$. The only dependent of g is h that already has $kc(h) = 1$. The new value of the Knuth counter for h is set to be $max(kc(h), kc(g) + 1) = max(1, 1) = 1$. Node h is already in sequence B and so is not appended to the end of this sequence during this iteration.

$n = 3$ - $head(B) = i$. There are no dependents for i so sequence $C = ()$, the empty sequence. Step 3 is not executed on this iteration and as a result and the algorithm proceeds to setting B to the sequence that is the tail of B removing the head element i from B.

$n = 4$ - $head(B) = e$. The value of the Knuth Counter for e is $kc(e) = 1$ and for its dependent g is $kc(g) = 0$. The new counter value for g is set to be $max(kc(e) + 1, kc(g)) = max(2, 1) = 2$. Element g is appended to sequence B and the head element e is removed.

The next five iterations of phase 2 of the algorithm are shown in Figure 4.6 for values $n = 5$ up to $n = 9$.

$n = 5$ - $head(B) = h$. This is the second time that h has been the node that is at the head of B in an iteration of the phase. The only node that is dependent

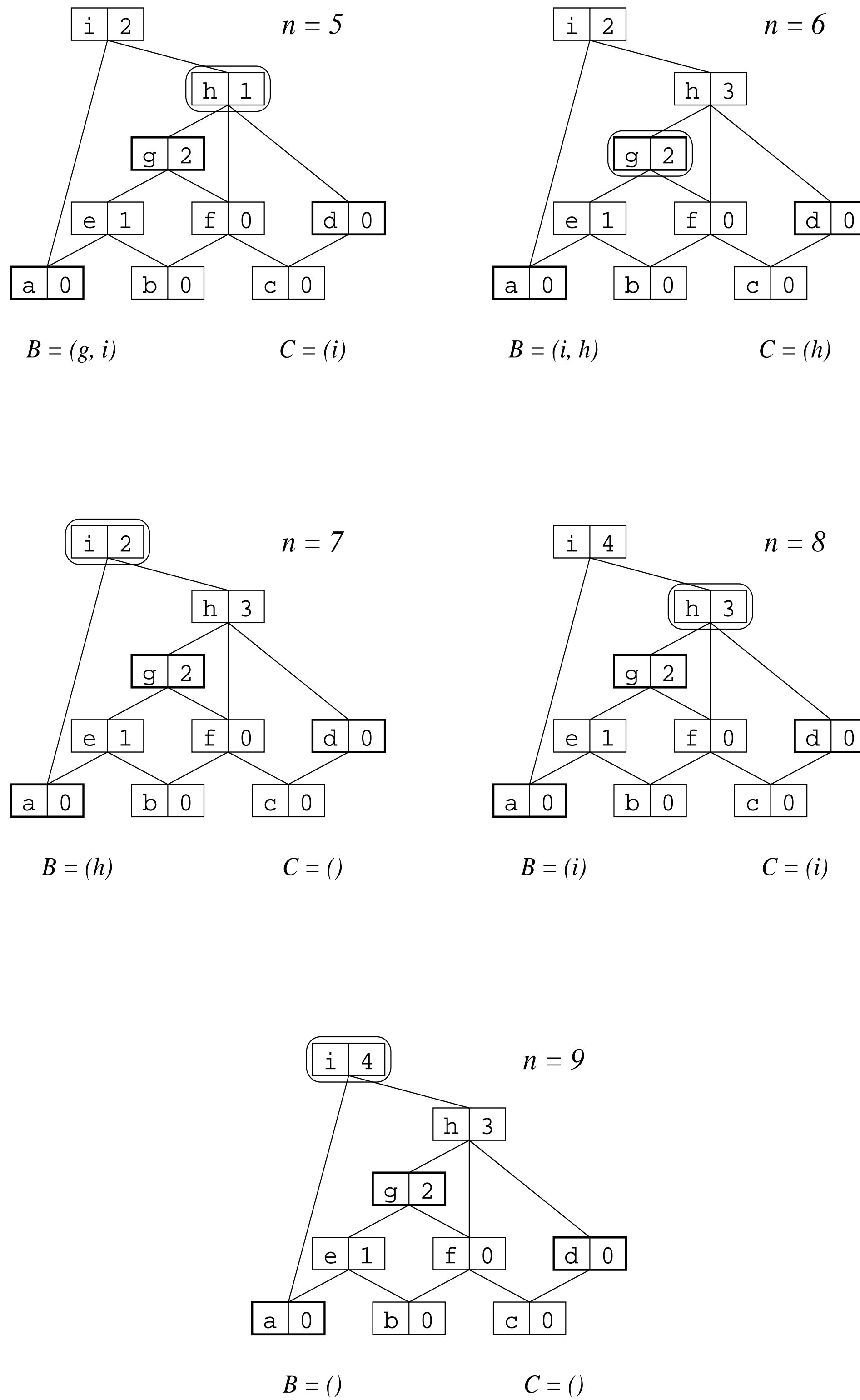


Figure 4.6: Intermediate states after n-th iteration in phase 2 of the block redefinition algorithm ($5 \leq n \leq 9$).

Iteration	Sequence B	Sequence C	Updates
initial	(a, d, g)		$\downarrow$ end of step 2
$n = 0$	(a, d, β')	(i, e)	$V(a) := value_2()$
$n = 1$	(d, β', e)	(h)	$V(d) := double(V(c))$
$n = 2$	(β', e)		—
$n = 3$	(e)	(g)	$V(e) := add(V(a), V(b))$
$n = 4$	(g)	(h)	$V(g) := add(V(e), V(f))$
$n = 5$	(h)	(i)	$V(h) := max_3(V(g), V(f), V(d))$
$n = 6$	(i)	$()$	$V(i) := power(V(h), V(a))$

Table 4.6: Phase 3 of the block redefinition algorithm.

on h is i and as $kc(h) = 1$ so the new counter for i is set to $kc(h) + 1 = 2$. B already contains element i.

$n = 6$ - $head(B) = g$. The Knuth counter for its direct dependent h is set to $kc(g) + 1 = 3$ and h is appended to sequence B.

$n = 7$ - $head(B) = i$ and as there are no dependents for i, step 3 is not executed and i is removed from B.

$n = 8$ - $head(B) = h$ for the third time in the phase. The counter for i is updated to be $kc(h) + 1 = 4$.

$n = 9$ - $head(B) = i$. As for $n = 7$ node i has no dependents so step 3 is not executed and i is removed from the head of B. The length of B is zero so the algorithm proceeds to phase 3.

At the end of phase 2, the topological sorting for the ordering of updates in complete. All nodes with a Knuth counter of zero and a thick border should be updated first, followed by any vertex with a Knuth counter of 1, then 2, and so on. This update is carried out in step 6 of phase 3.

Intermediate states during phase 3 are represented in Table 4.6. Each row in the table corresponds to an iteration. The values of sequences B and C in each row are snapshots of their values after step 3 of phase 3 in each iteration. The *Updates*

column shows the order of updates of values in V that need to be performed so that $\mathcal{M}_{S'}$ reaches a stable state, in which each value is consistent with its associated definition.

In place of a step-by-step description of the phase, it is sufficient to highlight its significant features. Although the reference g is associated with a redefinition in the block K, it is not appropriate to update its value before the update of a and e are completed. Nodes such as g have non-zero Knuth counters are replaced by a marker (β') in step 2 of the phase.

The execution of phase 3 then iterates between steps 3 and 9. Sequence C contains all the direct dependents for the current focus of the iteration, vertex $head(B)$. For any iteration n with focus node $a = head(B)$, each element x from sequence C that has a Knuth counter of $kc(a)+1$ is appended to the end of sequence B. This is the case only if x, which is dependent on a through being a member of sequence C, can be directly updated after the update of a. There should be no dependents of other vertices to take into consideration prior to this update. At the end of each iteration, the head of sequence B is removed.

The termination condition for the phase and the algorithm is that the sequence B is empty. This occurs when the final node to be updated in the phase has an empty sequence C.

4.4 Issues of Complexity with DM Models

Given the abstract DM Model for dependency maintenance and the algorithm for its update, it is possible to define a measure of complexity by counting the number of updates of values[6] required for a particular dependency structure. Section 4.4.1 describes some simple characteristics of the measure r_S with reference to illustra-

[6]Updates count recalculations and the setting of values that have no associated dependencies.

tive examples. Two more elaborate case-studies are presented in which well-known sequential algorithms are represented in a dependency structure. The first of these maintains the maximum and minimum of a set of values (Section 4.4.2); the second maintains a sorted sequence of numbers (Section 4.4.3). In these studies, the number of updates required to bring a DM Model up-to-date are tabulated.

4.4.1 A Complexity Measure for Dependency Structure

To be able to analyse the efficiency of the execution of scripts from their DM Model representation, there needs to be a method for counting the optimal number of updates that need to be performed for one state transition. This is analogous to counting the number of a particular type of operations (multiplication, compare/exchange etc.) are carried out in a conventional algorithm. This section focuses on how dependency structures influence the maximum number of updates required to carry out a single redefinition in a DM Model state.

Given a DM Model $\mathcal{M}_S$ with $n = \beta \Leftrightarrow \alpha$ vertices in its associated dependency structure, the measure r_S (introduced in Section 4.3.2) is an upper bound on the number of updates needed to update V via a single redefinition[7].

Four scripts Q, R, S and T, each with associated dependency structures with four vertices ($n = 4$), are shown in Figure 4.7. This figure illustrates the variety of dependency structures that can exist for a small value of n. For instance, the dependency structure may not be a tree nor be connected.

For each of the dependency structures in the figure, a solid dot represents a vertex in the structure that, if redefined, would require the maximum possible number of updates of values in one state transition. Note how the value of r for a structure depends on both the height and width of the structure.

Figure 4.8 sheds further light on the characteristics of the measure r_S. The

[7] A block of redefinitions K with size $\| K \| = 1$.

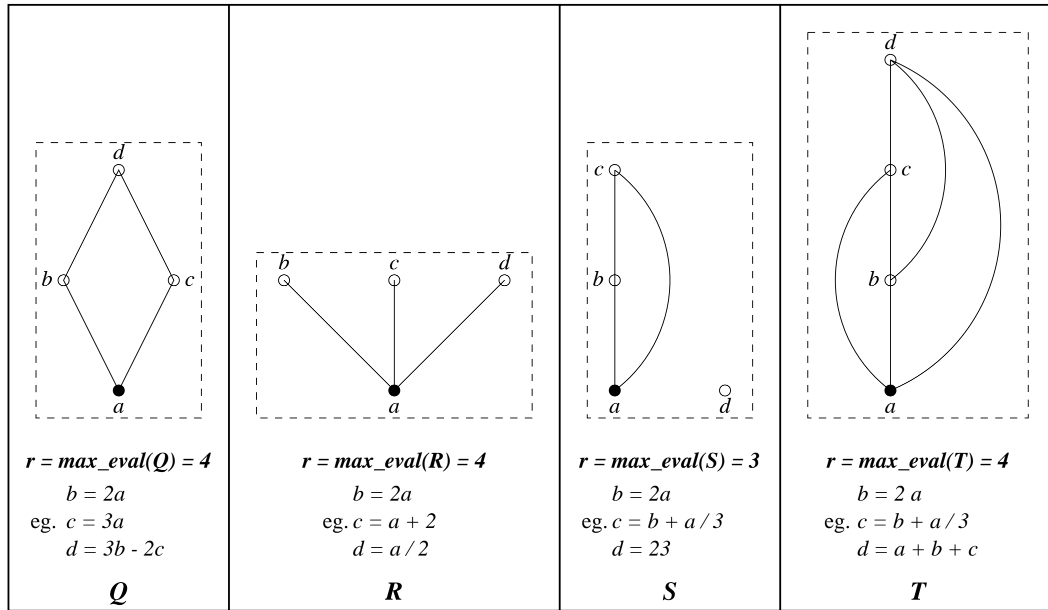


Figure 4.7: Possible patterns of dependency for four vertices.

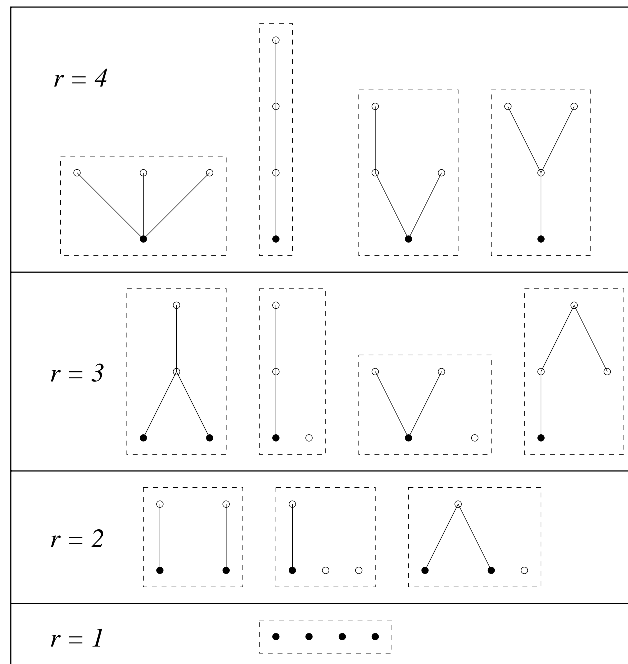


Figure 4.8: All possible patterns of dependencies for scripts with four definitions.

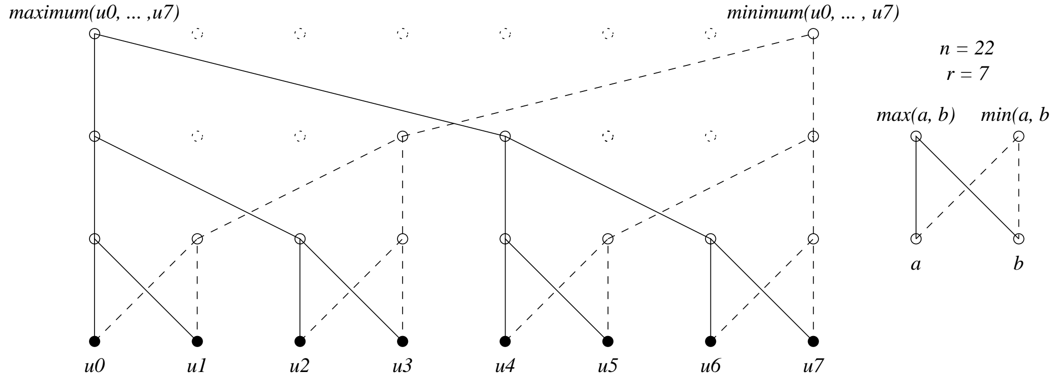


Figure 4.9: Dependency structure for finding minimum and maximum.

figure shows all the possible topologies for dependency structures associated with a script with four definitions, in which there is at most one path between each pair of vertices. As in Figure 4.7, vertices whose redefinition invokes the maximum number of updates are shown as filled circles. Future research will be directed at developing a theoretical framework for studying the measure r and associated dependency structures.

4.4.2 Case-study: Structure for Minimum and Maximum

This section examines a dependency structure developed to represent the dependency between a set of data and its minimum and maximum elements. It is appropriate to use the framework of the DM Model for this purpose, since the task of finding the maximum and minimum element of a set in a high-level definitive script with compound data types for values can be transformed to the same task with integers in a DM Model representable script. (As an application of such a structure consider determining the maximum dimensions of an enclosing bounding box for a group of geometric shapes as specified in a definitive script.)

Figure 4.9 shows a dependency structure that is associated with a script of definitions S that finds the maximum and minimum value of a set of integer values $\{u_0, \ldots, u_7\}$. The DM Model of script S has a maximum number of updates for one

Number of redefinitions	Minimum updates	Maximum updates	Minimum comp./exch.	Maximum comp./exch.
1	7	7	3	3
2	8	12	3	5
3	13	15	5	6
4	14	18	5	7
5	17	19	6	7
6	18	20	6	7
7	21	21	7	7
8	22	22	7	7

Table 4.7: Bounds for number of updates and compare/exchange operations for redefinitions - finding minimum and maximum value of a set.

redefinition of $r_S = 7$ in a structure with $n = 22$ vertices. In general, for a set of values $\{u_0, \ldots, u_m\}$ with script S_m for finding the minimum and maximum values (m is a power of 2), the maximum number of updates for one redefinition is r_{S_m}, where

$$r_{S_m} = 2 \log_2 m + 1 \tag{4.23}$$

The number of vertices n_{S_m} in the dependency structure for S_m in this general case is

$$n_{S_m} = 3m \Leftrightarrow 2 \tag{4.24}$$

A conventional algorithm for finding minimum and maximum values requires a minimum of $m \Leftrightarrow 1$ compare/exchange operations. To update the values of a dependency structure when all m values are modified takes $2m \Leftrightarrow 2$ updates. These updates can be paired into $n \Leftrightarrow 1$ *min* and *max* operations that are similar to the conventional compare/exchange operations. If the minimum and maximum values are initially up-to-date, then if only one value is altered through redefinition of an element of the set $\{u_0, \ldots, u_{m-1}\}$, the block redefinition algorithm would update r_{S_m} values. This is the equivalent of $\log_2 m$ compare/exchange operations.

Table 4.7 shows the bounds for the number of updates for finding minimum and maximum. Each row corresponds to the size of the block of redefinitions for

elements in the set $\{u_0, \ldots, u_7\}$. This table is constructed for the dependency structure depicted in Figure 4.9. The bounds for the minimum and maximum number of compare/exchange operations that the updates correspond to are also shown. The table shows that, when the block of redefinitions is smaller than the entire set, the block redefinition algorithm requires fewer updates than are required by the conventional algorithm for finding the minimum and maximum values for the whole set.

4.4.3 Case-study: Structure for Sorting

A similar analysis to finding the minimum and maximum of a set of values can be applied to sorting. A sorted sequence of values can be observed to be dependent on an unsorted set of values. Figure 4.10 shows a dependency structure that represents a script of definitions S for the Batcher bitonic merge sort [GS93] for a sequence of eight values $(u_0, \ldots, u_7)$. The sequence of sorted values is $(s_0, \ldots, s_7)$, where s_0 is the minimum and s_7 is the maximum value from the original sequence. If there is one redefinition of a value in the sequence then the upper bound for the number of updates required is $r_S = 35$.

In general, the number of compare/exchange operations required to sort a sequence of length m (a power of 2) using a Batcher bitonic sorting network is $m \log_2 m$. The total number of updates required to sort a sequence in a dependency structure is $2m \log_2 m + m$. This is the same as the number of vertices in the dependency structure.

Table 4.8 shows the maximum and minimum bounds for the number of updates and compare/exchange operations for the sorting structure illustrated in Figure 4.10, using the block redefinition algorithm. The updates are tabulated by row with reference to the size of the block of redefinitions $\| K \|$.

For small values of $\| K \|$, fewer updates and compare/exchange operations

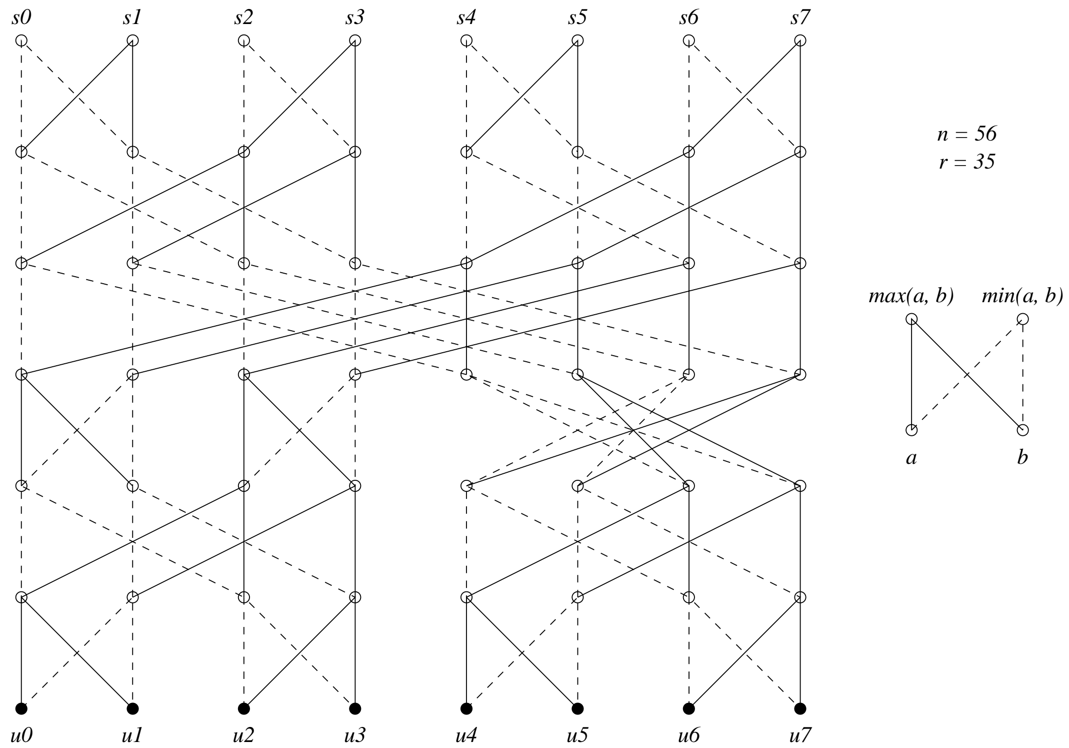


Figure 4.10: Dependency structure for sorting.

Number of redefinitions	Minimum updates	Maximum updates	Minimum comp./exch.	Maximum comp./exch.
1	35	35	15	15
2	36	46	15	22
3	39	49	16	23
4	40	52	16	24
5	51	53	23	24
6	52	54	23	24
7	55	55	24	24
8	56	56	24	24

Table 4.8: Bounds for number of updates and compare/exchange operations for redefinitions - sorting a sequence of values.

are required to sort the sequence of values. In the internal state of the DM Model, the intermediate values between the unsorted and sorted sequence are stored. Provided that the compare/exchange operations are computationally expensive relative to the ordering activity performed by the block redefinition algorithm, the definitive programming approach will be more efficient than a conventional procedural sort. Modulo the ordering activity, the block redefinition algorithm for dependency structure is always as least as efficient as the complete procedural sort.

The two case-studies demonstrate how dependency structure can be used as a form of algorithm. In the same way that the order of procedural execution of an algorithm can be optimised, the choice of dependency structure can affect the efficiency of state transition. The Batcher bitonic merge sort is chosen for the example as it is close to optimal for sorting. If bubble sort is used for the construction of a sorting dependency structure, then the number of updates required to complete the sort increases. Initial investigations into the representation of algorithms by dependency structure suggest that algorithms that are well-adapted for parallelisation are best suited for conversion to scripts of definitions. There are also clear connections between maintenance of dependency structures and dynamic algorithms.