ELECTRONIC PUBLISHING, VOL. 2(4), 231–256 (DECEMBER 1989)

Some computational properties of a model for electronic documents

TREVOR J. M. BENCH-CAPON AND PAUL E. DUNNE

Department of Computer Science
University of Liverpool
Liverpool L69 3BX, UK

SUMMARY

Differing types of documents exhibit varying structures. These may arise, at one level, because of the material comprising the text – thus textbooks will be organized differently from research papers – and at a lower level as a result of the layout conventions by which the text is formatted. These structuring regimes may be seen as defining a set of constraints which a document within a specific class must satisfy. In this paper we examine the model recently proposed in Reference [1] which is used for representing and modifying electronic documents. This employs simple graph grammars as a means of translating changes in the document structure into modifications to the computer representation. The aim of this approach is to provide computer support which will allow the appropriate structural conventions to be preserved while the document is being edited. We consider the following problem with this method: given a set of 'constraints' which the document must satisfy and a collection of rules prescribing how the document representation may be modified, how does one prove that *only* documents which obey the constraints can be generated by repeated applications of the rules? We describe one way in which this question can be more precisely formulated and call this the *consistency checking* problem. It is shown that, in general, this problem cannot be solved. We then outline how, for practical applications, the consistency checking problem may be solved for certain special cases.

KEY WORDS Document models Graph modification systems Electronic documents

1. INTRODUCTION

Any meaningful document has a structure, an understanding of which is essential if a reader is to interpret the text correctly. Over the years conventions for different classes of documents have developed; thus a textbook will typically comprise a preface, followed by a table of contents, followed by a series of chapters, followed by an index. Chapters themselves have structure, being broken down into sections, paragraphs and sentences. In the same way a mathematical paper is usually organized into: an abstract summarizing the content; an introduction describing the context in which the paper is set and outlining the results proved therein with possibly an informal description of the proof techniques employed; a list of definitions and notational symbols used; statements and proofs of preliminary results; statements and proofs of the main results; conclusions; and references. Normally the reader will be expected to read such documents in a particular order: therefore later sections may presuppose material in earlier sections. Understanding of the document is enhanced if the reader is aware of the structural conventions, but this means in turn that the author must be aware of, and observe, these conventions.

0894–3982/89/040231–26$18.00
© 1989 by John Wiley & Sons, Ltd.
Received 22 October 1989
Revised 5 February 1990

In recent years the use of computer systems as a mechanism for preparing documents has increased enormously, even though the final publication form will be as a hard copy. In such an environment the author interacts with a computer representation of the text (which we will subsequently call an *electronic document*) and this may have a radically different structure from that of the intended document. The simplest example of this is a standard text editor: here the computer representation of the document is just as a finite string of ASCII characters without regard to any higher level structure that may be present. The electronic document on which the author is working may have a different set of structural conventions from the intended finished product. In this way there exist two different views of a document: the author's conception of the structural organization of the text; and the model of the document used by the computer system. In a recent paper Furuta [2] calls the latter the *abstract representation* of a document and discusses the problems arising in transforming this, by stages called the *physical* and *page* representations, to a hard-copy form. In this paper we shall use the term *conceptual representation* to denote the high-level structural view of a document as seen by its authors and intended readership and are concerned with abstract representations for electronic documents which support this conceptual representation. We note that the conceptual representation may embody formatting constraints, e.g. the instructions for preparing manuscripts to appear in a conference proceedings might specify a maximum number of words, and that a particular point size, font and line spacing be used. Furuta, in Reference [2] notes that it is important to have a well-defined relationship between the abstract, physical and page representations in order to facilitate the process of producing the final version. Similarly Koo, in Reference [1], argues that the conceptual and abstract representations must also be closely linked: it is noted that document management systems must have some characterization of the documents dealt with and that how this characterization is made is a significant factor in the performance of the system with respect to such criteria as functionality and extensibility. That the conceptual and abstract representations be similar is also important in the field of collaborative authorship: suppose a number of writers have agreed upon the overall structure of a document so that there is a single conceptual representation; unless the abstract representation matches this it may be possible for one author to distort the structure of the final document. For example, consider a team of writers producing a user manual for a large item of software, each writer being responsible for describing some feature of the system. The team must decide on how much space to dedicate to documenting each feature, they must agree upon one set of notational and terminological conventions and fix a standard layout for the manual. If the abstract representation does not support the structure decided upon, then it is possible that one writer may expand the description of some facility beyond the space allocated for it, may use different notation and terminology, or may fail to respect the formatting style. While it is possible to recover from such deviations this would entail an unnecessary delay in the production of the final manual.

Use of directed graphs

One approach which has been widely studied as a means of linking conceptual and abstract representations is the use of directed graphs as a document model. These

provide a formal structure with which to represent documents in a computer system. We will use the term *document graph* to refer to a graph-theoretic representation of a document. In such models a document is viewed as a collection of (textual) *objects*: a node of the graph corresponds to a particular object and edges in the graph describe logical relations between objects, e.g. that a particular section must precede another section in the hard-copy form, that two sections are related by one being a statement of a theorem and the other being its proof etc. Nodes may also be labelled to describe the function played by the corresponding object in the text, e.g. that the object contains the text of an abstract, is a section title, is a definition etc. The use of node labelling permits sections of the document to be compressed: a node which is labelled as a *table* can subsequently be expanded into a subgraph which describes the table in terms of nodes labelled as table entries. This technique is appealing since it permits logical connections between sections of a document to be represented simply and directly. In addition various *attributes* may be associated with nodes of the document graph to indicate formatting conventions, e.g. that a particular font or point size should be used in setting the corresponding text. Examples of such models may be found in approaches such as hypertext [3–5] in which documents are represented by arbitrary directed graphs; in References [1, 6–8] where directed *acyclic* graph structures are used; and in References [9–12] which are restricted to tree structures.

However, since the document graph does not remain static but may alter as changes are made to the text, there is the possibility that a graph, which initially meets the constraints describing the logical structure of a document in a specific class, will cease to satisfy these after several modifications have been made. As a possible solution to this problem Reference [1] extends the earlier models of References [6–8] by introducing (amongst other ideas) the concept of *graph modification rules*. These rules, which are formally production rules of a graph grammar (see Reference[13]), are employed to control modifications to a document graph in order that it should reflect changes (either in structure or interpretation) made to the underlying document: thus rules may encapsulate how to modify the graph in the event of sections being added to or deleted from the document or rules may indicate how new logical links in the document structure are to be reflected in the document graph form. Koo's article illustrates how a simple set of rules may be applied to create and modify tables of information and he observes that the correctness of the given rules may be formally verified by a simple inductive proof.

The Koo model

In simple terms Koo's model of electronic documents involves the following elements: a specification of the class of documents being manipulated (e.g. a description of what constitutes a table possibly involving various constraints being placed on the data contained in the table); a collection of *initial* document graphs which will be expanded into full documents by the author (e.g. the initial graph for a model of tables could just be a single node labelled **table**); and a set of modification rules which are the means by which the initial and subsequent document graphs are modified (e.g. for tabular data such rules would describe the changes made to the graph when new rows and cells are added to a current table). A more formal description of this model will be given in the next section of this paper.

Following the approach in Reference [1], we shall in this paper assume document graphs to be acyclic. The properties of directed acyclic graphs offer several advantages over more general document representation methods particularly when their semantic capabilities are enhanced by the provision of a formal modification regime. It is important to note that by 'semantic' in this context we are referring to the *structural organization* of a document and *not* to its literal textual content. Thus in terms of the textbook organization described above the mapping of such onto a graph-theoretic formalism would encapsulate the high-level semantic view of this as: table of contents; sequence of chapters; index. However, our graph-theoretic model does not extend to a consideration of the actual words and punctuation contained in these components. To understand this distinction between what we shall call the high-level structural semantics of a document and the low-level textual semantics of the same document, ultimately just a finite sequence of ASCII characters, the following analogies may prove helpful.

Three examples

Classical imperative programming languages view programs as sequences of statements describing the precise actions to be performed. The order of the lines within this sequence is vital: different orderings will give rise to different program behaviours. The creators of such programs are normally not made aware of this implicit structure when writing a program since standard text editors handle programs and other documents in an identical manner: as a finite string of ASCII characters. Therefore it is entirely up to the program author to recognize the true program structure and to impose it on the document by physically inserting lines in the appropriate place. Similarly if the contributions of two or more authors are merged into a single program this must be achieved by laborious interleaving of the various contributions. Certain programming languages, provide the means to alleviate such difficulties: program statements are labelled with numbers; these numbers are assigned by the programmer(s) and the order of execution is determined by increasing values of the associated labels and not by the physical location of lines within the file. Now suppose we consider a program in such a language as a document and examine the creation and development of this. Then we have (at least) two different possible perspectives with which to analyse the semantic structure present: either that we view the document (program) created, as a block of text by the author, solely in terms of its *physical* representation as an ASCII character file within the computer system — this is the low-level description; or we take the viewpoint, implicitly adopted by the interpreter or compiler of the program, that the document is a sequence of labelled instructions with this sequence proceeding from the lowest numbered statement to the highest numbered statement — this is the high-level description. Note that the writer(s) of such a program must be aware that the high-level description exists and must construct their program accordingly in order to ensure its correctness: the individual statements comprising the code must be assigned numbers in accordance with their intended order of execution. On the other hand the programmer does not have to worry about physically reproducing this order in the program text: statements can be typed in any order provided the numbering convention is correct, so to insert a new statement between two existing ones it

would be sufficient to add this to the *physical* end of the file containing the program provided that the label of the new statement fell between the numerical values of the two statements it was to be inserted among.

As a second example one can consider the design of VLSI circuits. The low-level internal representation of such a circuit will be as a collection of, typically, several hundred thousand geometrical objects (rectangles, circles, lines etc.) corresponding to different semiconducting materials on the physical chip surface. However, a circuit designer regards these from a much higher level: groups of objects form transistors, configurations of transistors are organized into logic gates; collections of logic gates define complex components such as adders, ALUs, memories, processors etc. Several VLSI CAD tools are designed to support a structured design philosophy which represents a chip design as a regular array of *cells*, a cell being either primitive (i.e. defined in terms of its elementary physical structure) or *composite* (i.e. defined in terms of other cells). In this way the high-level semantic viewpoint treats a chip design as an array of components each with a well-defined functional behaviour and relationship to other cells; the low-level semantic view defines only the geometric layout of the entire chip surface and deals with this in terms of the electrochemical properties governing the behaviour of the semiconducting materials used.

As a final example of the difference between low-level and high-level representations one might consider the treatment of abstract data types (ADTs) in various programming languages. An ADT is defined in two stages: by specifying how a data structure is to be built using the basic data structures provided by the language (e.g. a *list* might be represented as an array of elements); and secondly by defining the permissible operations on the data structure (e.g. creating a list; adding an element to the head of the list; removing the head; concatenating two lists etc.). In most programming languages, particularly Pascal, Modula and Algol68, the underlying form of a data structure is not hidden from the remainder of the program and thus the writer of a program can make use of this, for example by accessing an element in the middle of list by referring directly to an array element instead of achieving the same effect by use of the 'allowed' operations. In other words such languages do not hide the low-level programming semantics of ADTs and this can often lead to avoidable errors in programs. In contrast, modern programming language theory has resulted in the development of languages such as Liskov's CLU and Milner's ML: the ADT mechanisms provided by these are constructed so that new data types can be created, accessed and changed *solely* by use of the operations specified when a structure is defined. Attempts to manipulate a data structure without the permitted operations result in compile-time errors. Thus these languages provide a high-level semantic view with the details of how a data structure is built being hidden from the remainder of the program.

Directed acyclic graphs for document management

It is one of the main contentions of this paper that an appreciation of the high-level structural semantics underlying any document *is essential* in the realm of cooperative authorship ventures and the examples just given provide some support for this claim. If a number of people are cooperating in the production of a program each individual will be responsible for contributing some well-defined functional component to the

final solution; clearly individuals must use labels which are globally consistent for their own section and by employing a creative tool which supports the high-level program view posited above this can be easily achieved without each writer having to know the set of labels employed by co-authors: the program is modelled as a sequence of nodes — one for each contributor — which are ordered according to the placement of the relevant blocks within the entire program. Each programmer can then write his own section of the final program, using whatever labels he wishes, and the document management system controlling the graph-theoretic representation of the program can then simply scan over the final code, following the section ordering, and assign correctly ordered labels to each statement. For example this could be accomplished by adding a sufficiently large number to each statement label: e.g. 1000 to each label in the 'first' block; 2000 to each label in the 'second' block etc. Thus, provided the authors have agreed on a synoptic program structure at the outset, the program can be merged simply by concatenating files and relying on the system's model to sort out the sequence without any need for physical interleaving of lines. In this case the advantages of using a model of the document appropriate to the document type and built into the system, over the superficially less constrained model of the finite text string, are evident. Again in the chip design example, individuals can concentrate on the low-level design of the specific functional components within the composite global design: without any high-level view of the design either cooperative work is impossible or (what amounts to the same thing) the chip surface is partitioned into physical segments with each designer deciding the content of each part without regard to the activities of other designers. Similarly, in languages which strictly control how ADTs may be manipulated, programmers, working cooperatively, do not have to be concerned with how structures defined by other authors are formed.

In this paper we are largely concerned with developing an approach capable of supporting the creation of documents having a highly organized structure. However, while it is possible to ensure that a high-level semantic form is adhered to, it is virtually impossible to enforce correctness of the low-level textual semantics: e.g. in a mathematical paper in which (at a high level) a theorem statement should be followed by a theorem proof, the document management tool cannot ensure that the textual contents of the 'statement' block are in fact a theorem statement or that the textual content of the 'proof' block do indeed constitute a proof. Accepting this *caveat* we contend that directed acyclic graphs provide the strongest basis for a high-level document management tool.

One important feature of such graphs is that there are only a fixed number of finite paths permitted by the graph structure from source nodes to terminal points. This allows the sensible readings of the final document, when produced in an orthodox hard-copy version, to be reflected in the computer representation used for writing and revising the document. Here there is an underlying assumption that the sensible paths through any document are acyclic. Again it is stressed that this pertains to the high-level structural organization of a document and not to how it might be read when produced as hard copy: the prerequisite material for a chapter of a textbook must precede the chapter which assumes a knowledge of it (regardless of the fact that a reader may wish to pass through it more than once); the supporting definitions and lemmata for a mathematical theorem must be stated (and possibly

proved) prior to the announcement of the theorem itself. In fact one may reasonably contend that every document has a directed, acyclic high-level structure in this sense: textbooks, mathematical papers, user manuals, works of fiction (*pace* the exceptional case of Reference [14] which to treat as acyclic would be to misunderstand the author's aims).

These constraints are important in avoiding the confusion that can result from the more liberal representation methodologies such as hypertext techniques and, in particular, the semantic shifts and inversions that are characteristic of the hypertext approach can be prevented. A fuller discussion of such issues is the subject of Reference [15] Of course it might be contended that by restricting oneself to a subset of hypertext's capabilities one can achieve the same ends, and still have in reserve a battery of techniques for use with more general representations. However, to argue thus indicates a failure to appreciate the expressive clarity of minimalism: First-order Logic does not become a redundant theory even though Second-order Logic provides greater mathematical potency; it is easier to prove properties of systems whose behaviour is rigidly controlled by a concise set of construction principles than it is to reason about the outcome of methods which permit great laxity in specification, even if the latter can be artificially restricted to emulate the former.[1] An additional advantage of the graph modification rule paradigm is that it provides a rigorous structure with which to regulate cooperative authorship ventures. The modification rules guarantee that, regardless of how many writers are working on a document and the specific changes they make to it, the global organization of the text will always conform to a predefined model so that the conventions of the target document are observed: while co-authors work on particular sections if changes are made in accordance with the modification rules then these will ensure that a homogeneous structure results. Note that this substantially reduces the requirement for multiple authors to expend considerable efforts in collating individual contributions.

There are two difficulties which arise in implementing the method described above: first how to construct the appropriate modification rules — ideally one would wish to do this automatically given some formal specification of the document class; secondly how to determine if the modification rules are correct in the following sense: the specification may be given by describing the properties that any document graph (the abstract representation) must possess in order for it to correspond to a document in the class considered (the conceptual representation); as a minimal requirement the modification rules when applied repeatedly to the initial graphs should generate only document graphs which satisfy the specification conditions. We shall call a set of modification rules which exhibits this behaviour *consistent* with respect to the given specification.[2] Thus the second difficulty present is how to ensure that the modification rules are consistent with the specification of the

[1] The ADT example is particularly instructive with regard to these points: in principle one might propose a methodology for Pascal program development which would assert the necessity of manipulating user-defined data structures solely by means of the operations specified for them. In practice one could not *enforce* the use of such an approach without making corresponding changes to the Pascal language definition (and then, of course, one is no longer dealing with Pascal *per se*). In short, when restricting some general set of techniques to a subset of the same, it is impossible to prevent the conversant user from moving outside the intended regime and utilizing facilities which were intended to be hidden, e.g. accessing a list element via an array reference.

[2] In Reference [1] Koo defines consistency in a rather narrower sense: a set of rules is consistent if it generates only graphs which are 'well-formed'; since the paper requires document graphs to be acyclic, directed graphs containing cycles are not regarded as well-formed.

document class: i.e. the abstract representations always agree with the conceptual representations.

One of the results proved in this paper is that the problem of checking inconsistency is undecidable. The proof involves a reduction from the Halting Problem for Turing machines. As a consequence of the proof technique it turns out that graph modification rules may be viewed as a formal model of computation. This viewpoint may be used to establish that certain restricted variants of the inconsistency checking problem, although decidable, are still intractable.

In Sections 2 and 3 we are concerned with results of a theoretical nature: Section 2 reviews the formal definitions of document graph and graph modification rule as given in Reference [1] and gives a precise formulation of the consistency checking problem. In Section 3 we show that no general method exists for solving the consistency checking problem. In that section we show that, despite the theoretical impossibility of checking consistency in general, for a number of practical cases, specifications of particular document classes can be defined and a set of consistent modification rules constructed for them. Then Section 4 presents modification systems capable of describing abstracts, documents containing tables, and mathematical papers. It should be noted that here we are concerned with depicting the logical organization of such documents by specific systems of modification rules. The issue of designing graph modification systems dealing with structural aspects (in particular text-formatting conventions) will be dealt with in a subsequent paper.

2. DOCUMENT GRAPHS AND GRAPH MODIFICATION RULES

In this section we set up formal definitions of the concepts introduced in the introduction. These definitions will allow us to state precisely what it means for a rule system to be consistent with respect to a specification.

Definition 1: ([1]) A *document graph* is a directed acyclic graph, $G(V, E)$. The vertices in V denote *objects* in a document and the edges in E depict logical connections between objects. Each object has an associated *object type*. This consists of two parts: a *data type* which specifies the domain of possible data values for the object (e.g. integers, text strings); and an *attribute type* which indicates the domain of possible properties that the object may possess (e.g. font, size). Objects may also be labelled. For a fuller analysis of object types the reader is referred to Reference [1]. •

Document graphs are the *abstract representations* of document structure: the form that is manipulated by the author when editing the document within a management system. Recalling that our objective is to have this abstract representation closely matching the author's conceptual representation the following definition provides a method of specifying which documents belong to a particular class in terms of properties of the graphs representing them.

Definition 2: A *document specification* consists of a pair $DS = (C, Init)$. Here C is a finite set of *constraints*,

$$C = \{ C_1, C_2, \ldots, C_k \}$$

where each C_i is a (computable) predicate on document graphs. *Init* is a set of *initial* document graphs. Given a document specification DS and a document graph G, G is

said to *meet* the specification DS if and only if $G \in Init$ or $C_i(G)$ is **true** for each constraint C_i. ●

Note that the specification is defined in terms of document graphs and *not* documents themselves. In addition although we have formally defined constraints as a set of predicates in practice one would envisage the specification as being generated from some less rigorous description; e.g. our informal structure for mathematical papers, given in the introduction, could be translated into a set of conditions among which might be the predicate 'every node labelled **proof** has a unique predecessor which is labelled **theorem** or **lemma**'

Definition 3: A *graph modification system* (or GMS) is a finite set

$$S = \{R_1, R_2, \ldots, R_m\}$$

of *graph modification rules*. Each graph modification rule, R, is a triple $<P, G_l, G_r>$ where P is a predicate on document graphs and G_l, G_r are document graphs. A rule $R = <P, G_l, G_r>$ acts on a given document graph G as follows: if $P(G)$ is true and G contains G_l as a sub-document graph then G_l in G is replaced by the document graph G_r. In general applying a rule R to a document graph G results in a new graph H. We say that G *yields* H (denoted $G \rightarrow H$) in this case. Similarly if H results from repeated applications of rules to G we say that G *derives* H (denoted $G \rightarrow^* H$). ●

Now, given a document specification there will in general be infinitely many document graphs which meet that specification; similarly given a GMS acting on the initial graph of a specification there may be infinitely many new document graphs which can be derived by repeatedly applying the modification rules. In order for the GMS to be 'correct' each document graph which is derived using it should meet the specification. Using this approach we can now formally define the concept of *consistency*. Our definition subsumes the informal notion of consistency put forward in Reference [1].

Definition 4: Let $DS = (C, Init)$ be a document specification. $Good(DS)$ is the set of all document graphs which meet DS. Let S be a GMS. The *derivation set* of S is the set of graphs, $\Delta(S)$ defined by

$$\Delta(S) = \{H : \exists\, G \in Init \text{ such that } G \rightarrow^* H\}$$

S is *consistent with respect to* DS if and only if $\Delta(S) \subseteq Good(DS)$, i.e. every document graph derived using S meets the specification DS.

The *inconsistency problem*[3] for graph modification systems is the following:

The Inconsistency Problem

Input: $DS = (C, Init)$ a document specification;

$S = \{R_1, \ldots, R_m\}$ a GMS operating on the initial graphs of DS.

Output: True if S is inconsistent with respect to DS, i.e. there is a graph in the derivation set of S which does not meet the specification DS. False otherwise, i.e. if S is consistent with respect to DS.

3. THE UNDECIDABILITY OF THE INCONSISTENCY PROBLEM

We begin by recalling a few elementary definitions from the field of computability

[3] The reasons for phrasing the problem as that of checking inconsistency rather than consistency will become apparent below.

theory. For further details and technical proofs the reader is referred to Reference [16].

Definition 5: A *Turing machine* consists of a two-way infinite tape divided into cells which are numbered $\ldots, -3, -2, -1, 0, 1, 2, \cdots$; a read/write head which can scan a single cell of the tape at a time and move one cell left or right after each computation step; and a processor which controls the head actions according to the current state and symbol being read. At any time each cell contains exactly one of the symbols in $\{0, 1, B\}$, where B is a special *blank* symbol. The processor at any time is in exactly one state, q, taken from a finite set of states, Q. Three special states in Q are identified: q_0, the *initial* state which is the state of the processor at the start of any computation; q_A, the *accept* state: if this is reached then further computation halts and the result *True* is returned; q_R, the *reject* state: if this is reached then computation ceases and the result *False* is returned. The action of a Turing machine program, M, is completely described by the *state transition function*

$$\delta : Q \times \{0, 1, B\} \rightarrow Q \times \{0, 1\} \times \{L, R\}$$

Suppose the processor is in state q_i and the head is scanning a cell containing the symbol $\alpha \in \{0, 1, B\}$, then $\delta(q_i, \alpha) = (q_j, \beta, D)$ for some $q_j \in Q$, $\beta \in \{0, 1\}$ and $D \in \{L, R\}$. In this case the head prints the symbol β in the cell it is scanning and moves one cell left (if $D = L$) or one cell right (if $D = R$); finally the state of the processor is changed to q_j.

Such a machine checks predicates as follows: initially the input to be tested (which is a string of n binary symbols) is placed in cells $1 \cdots n$ of the tape all other tape cells containing the B(lank) symbol. The tape head scans cell 1 and the processor is in state q_0. The processor modifies the tape and controls the tape head movement following the prescription of δ. Computation ceases if the processor enters state q_A or q_R. □

It is well known that there are predicates which cannot be solved by Turing machine programs guaranteed to halt for every possible input. Such problems are said to be *undecidable*. The following problem is an example.

Definition 6: The *Halting Problem* for Turing machines is defined by:

Input: Turing machine program, M, and an input x for M

Output: True if M when started with input x eventually halts; False if M never halts when started with input x. □

Theorem 1: ([17]) The Halting Problem for Turing machines is undecidable. □

We can now prove the main result of this section.

Theorem 2: The Inconsistency Problem is undecidable.

Proof: Given a Turing machine program, M, (for which it is sufficient to describe only the state transition function δ), and input data $x = x_1 x_2 \cdots x_n \in \{0, 1\}^n$ for M we describe how to construct a document specification, $DS(M, x)$, and a graph modification system, $S(M, x)$. The graph modification system will be inconsistent with respect to the specification if and only if M halts on input x. This proves the result since any effective procedure solving the Inconsistency Problem would yield an effective procedure solving the Halting Problem for Turing machines and Theorem 1 establishes that no such method exists.

The constraint set of $DS(M, x)$ contains just a single predicate, C, on directed graphs: $C(G)$ is **true** of a directed graph G if and only if G is acyclic. Similarly

$DS(M, x)$ contains exactly one initial graph, $G_0(x)$, which we describe below.

The idea behind the proof is to view the contents of the non-blank portion of the tape and the current state and head position as constituting a document. The graph modification rules then describe how this graph should be changed in accordance with steps carried out by the processor obeying the transition function. $G_0(x)$ thus corresponds to the starting configuration of M on input x. The objects in the document graphs to which the system $S(M, x)$ applies have attributes drawn from the set $\{0, 1, B\} \cup Q$ where Q is the set of states of the processor executing M. For all $t > 0$ the document graph G_t will be the *unique* graph yielded by the document graph G_{t-1}. G_t will contain precisely $m+3$ vertices, where m is the number of non-blank tape cells after t moves of M have been executed (thus G_0 contains $n+3$ nodes: n nodes of input, one node indicating the head position, and two nodes indicating blank cells on either side of the input data). In addition G_t has the following properties: There is a unique *source* node v_0 which has in-degree 0; a unique *sink* node v_{m+2} which has out-degree 0; both of these nodes have attribute B. The nodes v_i for $1 \le i \le m+1$ each have in-degree and out-degree equal to 1 and for all $0 \le i \le m+1$ (v_i, v_{i+1}) is an edge of G_t; no other edges are present. The attributes of each object in G_t will be constrained as follows: if after t moves the tape head is scanning the ith non-blank cell (from the left) and the processor state is q_j then the node v_i has the attribute q_j; the nodes v_k for $1 \le k < i$ have the attribute corresponding to the kth non-blank symbol on the tape at this time; similarly the nodes v_l for $i+1 \le l \le m+1$ have the attribute corresponding to the $(l-1)$th non-blank symbol currently on the tape. Below we say that a document graph, G, is a configuration graph for M if and only if: G contains p nodes connected in a linear arrangement; the first and last nodes each have attribute B; exactly one node has attribute q for some $q \in Q$; all remaining nodes v_k have attribute α_k for some $\alpha_k \in \{0, 1, B\}$. In the description which follows v_i for $0 \le i \le p-1$ denotes the nodes in a configuration graph in which (v_i, v_{i+1}) is an edge for each $0 \le i \le p-2$.

Thus if $x = 0011$ then $G_0(0011)$ will be the document graph depicted in Figure 1 below:

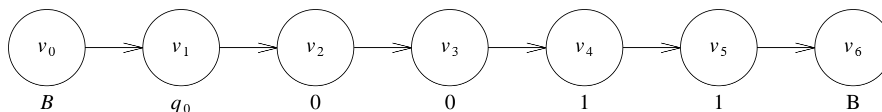

Figure 1

$S(M, x)$ contains a constructor rule (P, G_l, G_r) defined as follows: $P(G)$ is true if and only if G is the empty graph; G_l is the empty graph and G_r the graph $G_0(x)$ defined above. Note that $G_0(x)$ is a configuration graph for M.

For each move $\delta(q_i, \alpha) = (q_j, \beta, D)$ of the state transition function for M there will be 2 graph modification rules present in $S(M, x)$. These rules fall into one of two classes depending on whether $D = L$ or $D = R$. We describe these in turn:

Case 1: For moves $\delta(q_i, \alpha) = (q_j, \beta, L)$

Rule 1: $<P, G_l, G_r>$

$P(G)$ is true if and only if: G is a configuration graph for M, v_1 has attribute q_i; v_2 has attribute α. In this case G_l and G_r are the sub-graphs in Figure 2. Unlabelled vertices in G_r are new nodes.

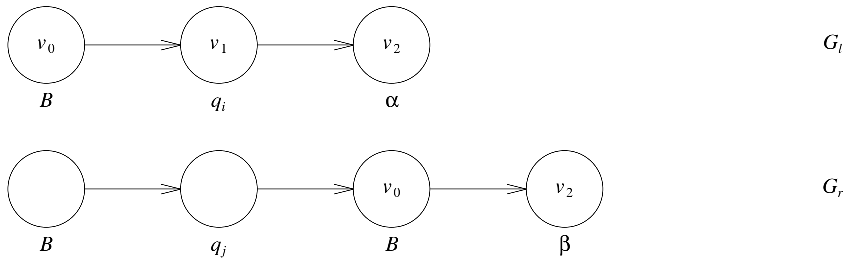

Figure 2

Rule 2: $< P, G_l, G_r >$
$P(G)$ is true if and only if: G is a configuration graph for M; there is a node v_k with attribute q_i; a node v_{k+1} with attribute α and $k \neq 1$ (hence $k \geq 2$ by definition of configuration graph). G_l and G_r are depicted in Figure 3.

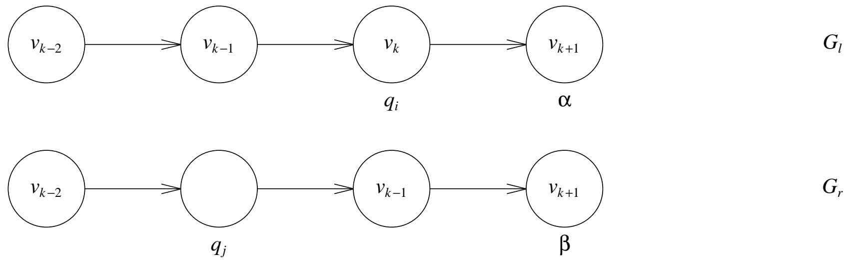

Figure 3

Case 2: For moves $\delta(q_i, \alpha) = (q_j, \beta, R)$
Rule 1: $< P, G_l, G_r >$
$P(G)$ is true if and only if: G is a configuration graph for M; for some node v_p, the out-degree of v_p equals 0; v_p has attribute α; the node v_{p-1} has attribute q_i. G_l and G_r for this case are shown in Figure 4.

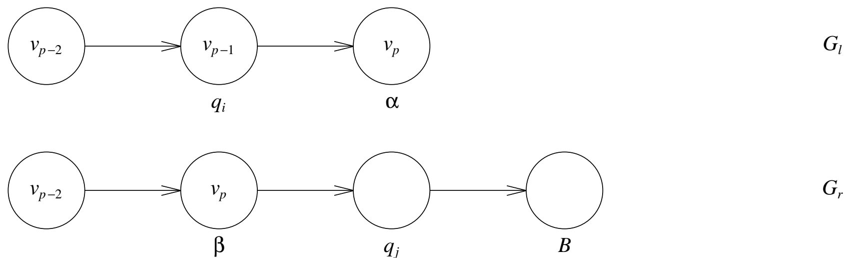

Figure 4

Rule 2: $< P, G_l, G_r >$
$P(G)$ is true if and only if: G is a configuration graph for M; there is a node v_k such that v_k has out-degree 1 and v_k has attribute α; the node v_{k-1} has attribute q_i. G_l and G_r for this case are shown in Figure 5.

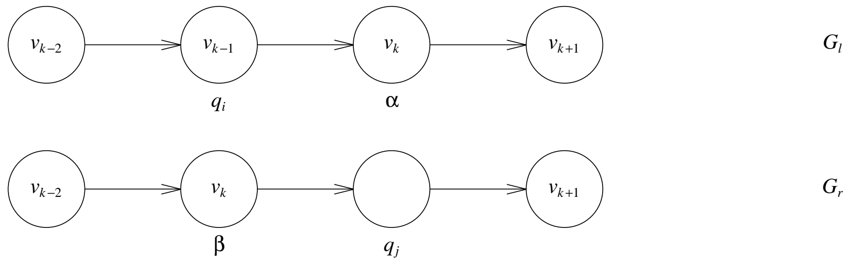

Figure 5

Finally there is a rule to handle termination.
Rule 3: $< P, G_l, G_r >$
$P(G)$ is true if and only if: G is a configuration graph for M and there is vertex v_k of G with attribute $q_F \in \{ q_A, q_R \} \subset Q$. In this case G_l and G_r are as Figure 6.

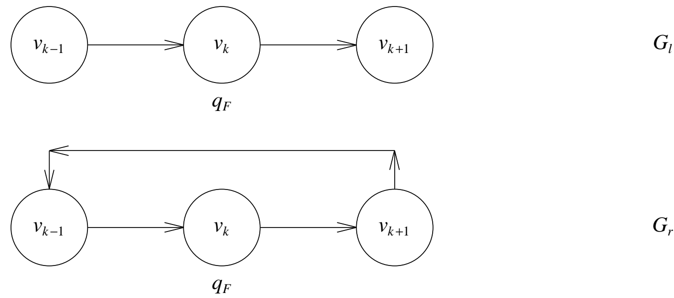

Figure 6

This completes the description of the graph modification system $S(M, x)$.

We claim that $S(M, x)$ derives a graph which is not a document graph from $G_0(x)$ if and only if M halts on input x. This follows easily from the observations below:

(1) If $G \in \Delta(S(M,x))$ is a document graph then there is a unique graph H such that $G \rightarrow H$ using $S(M, x)$. This holds since the graph modification system $S(M, x)$ is built on the state transition function so that exactly one modification rule is applicable to G.

(2) Let $G_0(x)$ be the graph derived from the empty graph using the constructor rule and $G_t(x)$ (for $t > 0$) denote the graph resulting after t applications of modification rules to $G_0(x)$. Then for all $t \geq 0$ G_t encodes the tape contents, head position and current state of the Turing machine M after t moves when started with input x. To see this note that the sequence $G_0, G_1, \ldots, G_t$ is uniquely defined from x and that the attributes of each node in G_t describe the tape contents, head position and current state of M is guaranteed by the modification rules.

(3) M halts on input x after t moves if and only if $G_{t+1}(x)$ contains a directed cycle. This follows since if M has halted after t moves then some vertex in G_t has attribute q_A or q_R. The termination rule of Figure 6 now produces a directed cycle in G_{t+1}.

In summary (3) states that M halts on x if and only if the GMS $S(M,x)$ is inconsistent with respect to $DS(M, x)$. Thus the Inconsistency Problem is undecidable. □

4. PROVABLY CONSISTENT GRAPH MODIFICATION SYSTEMS FOR SPECIFIC DOCUMENT CLASSES

Although the result of the previous section indicates that there can be no generally applicable method for checking the consistency of a rule set, in practice this need not cause any difficulty. The main reason for this is that one can attempt to build modification rules from the specification directly in such a way that that their consistency follows automatically. In the remainder of this paper we give informal specifications for three classes of document and describe modification systems for these which will be self-evidently consistent. A system for formally specifying documents and generating consistent rule sets automatically from these is currently being investigated by the authors and will be the subject of a subsequent paper.

A document graph might be used to perform two possible tasks: as a means of encoding the internal structure of a document for the purposes of storage, editing and eventual layout production; and secondly to provide a mechanism for readers to browse through a document to obtain information concerning its content. Although these two roles are not mutually exclusive — one may wish to provide facilities for readers to add information to or correct errors in a document while browsing it — the differing functional requirements in each case will affect the form of rules provided in the corresponding modification system. Thus in the former case modification rules are used to incorporate textual changes and, for formatting purposes, additional rules will be needed to deal with re-organizing paragraph and page layouts in the event of page- and line- length parameters, fonts, and hyphenation, widowing and orphaning conventions being altered. However, the representing graph structure would be in effect linear without explicit logical links between sections of text being recorded. In contrast, for browsing purposes, although some elementary editing functions would be included, the primary requirement of the graph will be to provide an effective mechanism for traversing the document content. So in this case rules will be present to indicate logical connections between related sections of a document. As mentioned above, our intention in this paper is to concentrate on the latter application.

The fact that document graphs are directed and acyclic implies certain constraints on how the represented document may be scanned according to the graph. So the following result summarizes well-known and easily proved properties of directed acyclic graphs.

Fact 1: In any directed acyclic graph:

(i) There is at least one node $v \in V$ which has no incoming edges.
(ii) There is at least one node $w \in V$ which has no outgoing edges.
(iii) A directed graph $G(V, E)$ is acyclic if and only if it may be *topologically ordered*, i.e. each node $v \in V$ may be assigned a unique integer label, $\lambda(v)$, with $1 \leq \lambda(v) \leq |V|$ so that for all edges (v, w) of G the relation $\lambda(v) < \lambda(w)$ holds. □

Property (i) implies that any document graph has some non-empty set of *entry points*: sections in the document from which a reader may start browsing. Similarly property (ii) implies that any document graph has a non-empty set of *terminal points*: places from which no other part of the document can be reached. Finally property (iii) shows that there are a limited number of defined paths through the document, these arising from any topological order. The simplest example of the browsing regime imposed by these properties is the UNIX† command **more**. In contrast the behaviour of the command **less** could not be simulated by a consistent graph modification system since it provides the facility to re-read portions of a document.

We now describe graph modification systems for some particular classes of document.

Abstracts and document headers

A document header may be specified as consisting of a *Title*, *Author list*, *Abstract* and *Keyword list* in that order. The *Title* and *Abstract* are unique (i.e. exactly one node with these attributes occurs) and only the *Keyword list* is allowed to be empty.

In describing a modification system for this class we wish to have available as editing facilities mechanisms for: creating the document graph of an abstract; adding or removing items from the author list; adding or removing items from the keyword list. In addition it must be possible to elicit complete information about the document from the structure of its graph.

We describe rules by giving the sub-graph G_l (which is to be replaced), the replacing graph G_r, and the predicate, P, which must be satisfied in order for the replacement to apply. In the diagrams below nodes with the same internal label are common to both G_l and G_r; nodes with no internal label in G_r are new nodes; finally the labels adjacent to nodes describe the node attribute.

Rule 1.1: Creating a header

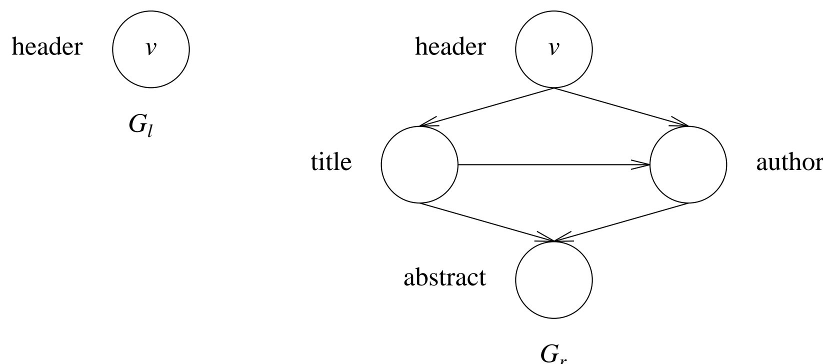

Figure 7

† UNIX is a trademark of Bell Laboratories.

The predicate associated with this rule is just the constant value **true**.

This organization permits a textual traversal starting from the header node through both *author* and *title* (or only one of these) and onto the descriptive *abstract*.

Rule 1.2: Adding an author
We give only the rule for appending a new author to the end of an author list. The rules for inserting into the middle or adding to the start may be derived in a straightforward manner from this.

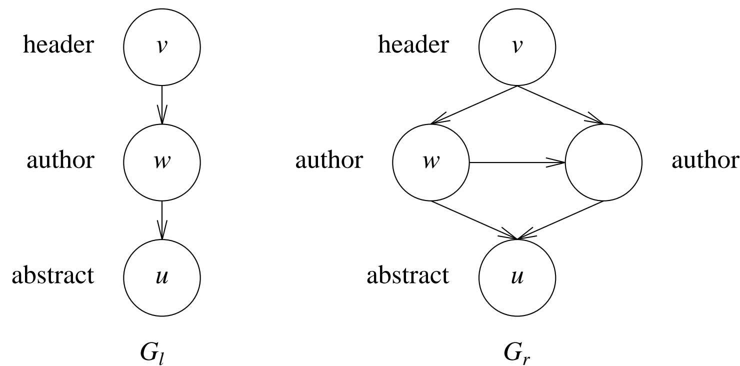

Figure 8

The predicate associated with this rule is **true** if and only if the node labelled w in G_l has exactly one outgoing edge (i.e. connecting it to the *abstract* node)

Rule 1.3: Deleting an author
These rules just reverse the processes of Rule 1.2.

Rule 1.4: Creating a keyword list

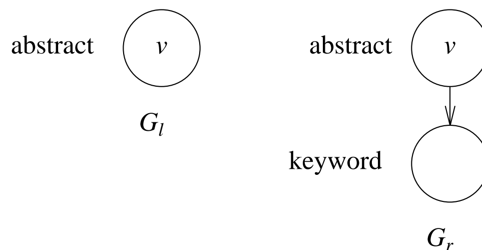

Figure 9

The predicate associated with this rule is **true** if and only if the *abstract* node, v, has no outgoing edges.

Rule 1.5: Adding a keyword
Again we give only the rule for appending a keyword to the end of the list.

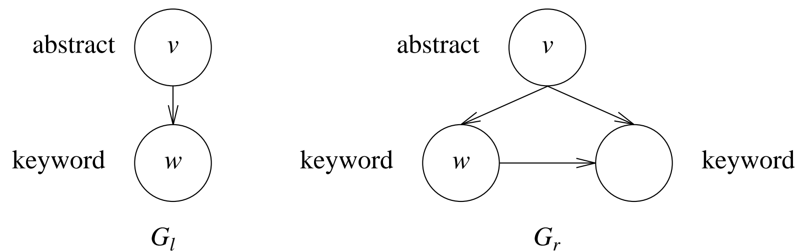

Figure 10

As with the rule for adding an author, the predicate, P, is **true** if and only if the node labelled w no outgoing edge.

Rule 1.6: Deleting a keyword
These merely reverse the processes of Rule 1.5 except in the case of there being just one keyword when the reverse of Rule 1.4 is applied.
Theorem 4: The graph modification scheme given by the rules (1.1)–(1.6) is consistent.
Proof: It suffices to show that any graph generated by the rules is acyclic. To see this observe that any generated graph may be partitioned into three or four levels corresponding to *header*, *title and authors*, *abstract* and an optional *keyword* section. The first and third of these just contain a single node, while the nodes in the second and fourth are linearly ordered from left to right. It follows that any such graph may be topologically ordered and hence from Fact(1)(iii) is acyclic. □

It is also easy to see that there is a directed path through any generated graph which contains every node, thus the rules create graphs which can be applied to completely scan the corresponding document.

Papers containing tabular information

Reference [1] gives a consistent modification system for maintaining tables and in the description of graphs representing papers below we shall assume this is the approach used for tabular data. As such we will not repeat the lengthy details of the rules given in Reference [1].

A paper may be seen as comprising a header (i.e. title, authors and abstract as above) followed by some non-zero number of sections which constitute the body of the paper. At this point we regard a section of a paper as comprising a sequence of *blocks*, a block being either a paragraph of text or a table. In principle different structural analyses are possible: one might take account of the subject matter of the paper which, as in the case of mathematical papers for example, may conform to a precisely ordered pattern; alternatively one could impose additional hierarchical levels by dividing into sub-sections etc.

The modification system below provides: rules for creating a paper; editing rules for adding and deleting sections or blocks; rules for adding forward reference links

between textual blocks and tables; and rules for adding *section omission* links. A node with attribute *header* in the graphs below denotes a document graph of the type described in Section 3 above augmented by an additional terminal node into which an edge from each keyword is directed (or from the abstract if there are no keywords). This convention allows us to represent the possibility of there being several different entry points into a document. Thus, for certain papers, specialists familiar with the background to the subject matter of a paper may wish to skip those introductory and definitional sections following the abstract and proceed directly to the technical exposition in the main body of the paper; other readers may only be interested in the abstract and introduction/conclusions. Note that below a section is regarded as having a *section header*, *section body*, and *section end* node in it. The attribute *section body* is again used as a shorthand to hide the document graph for a sequence of blocks. As previously the rules for deleting objects will not be stated explicitly since they are easily obtained by reversing the corresponding additive rules.

Rule 2.1: Creating a paper

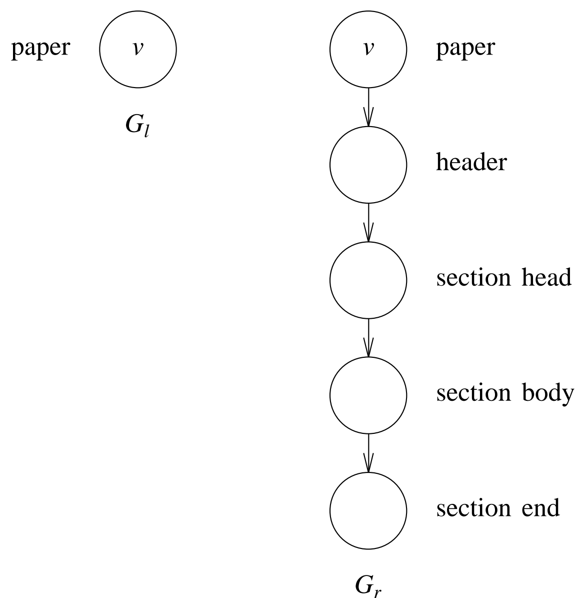

Figure 11

The predicate associated with this rule is just the constant value **true**. Note that the initial *section body* will just be a single block (paragraph or table).

Rule 2.2: Expanding a section body by adding a block
The unattributed nodes in the graphs below will in practice have attributes taken from: *section head*, *paragraph*, *table*, or *section end*. The legal combinations are covered in the predicate associated with this rule. The attribute *block* is either *paragraph* or *table*.

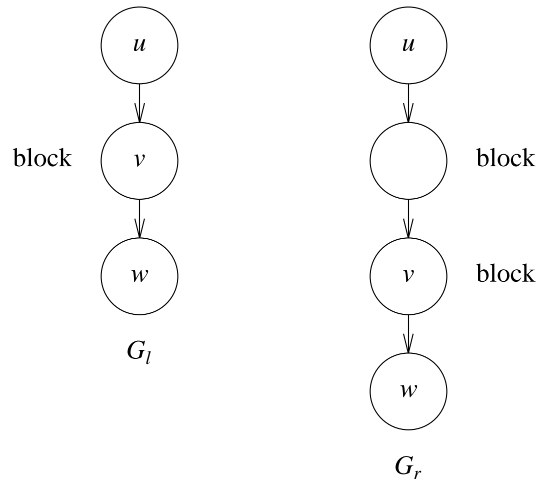

Figure 12

For this rule $P(G)$ is **true** if and only if the following condition holds:

u has attribute *section head* or *block* and w has attribute *block* or *section end*.

Rule 2.3: Adding a new section

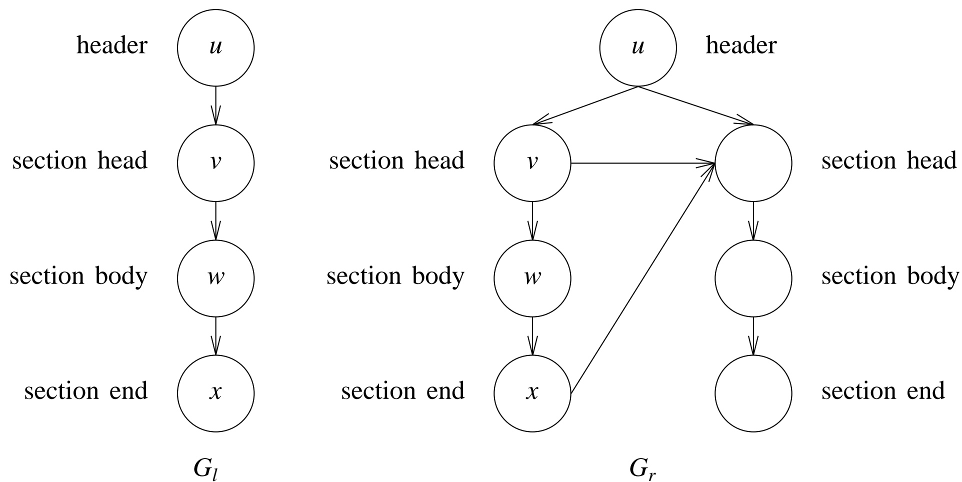

Figure 13

For this rule the associated predicate is **true** if and only if the *section head* node (v) has exactly one successor.

Again observe that a complete traversal of the text is possible but that if desired a particular section can be scanned directly without investigating its predecessors.

Rule 2.4: Adding a reference between two blocks
It is important to note that only *forward* references are permitted between blocks in the same section, e.g. there might be a link from a paragraph to a table but there would not be any link from the referred object to the referring text. This requirement is needed to preserve the acyclic nature of the graphs generated.

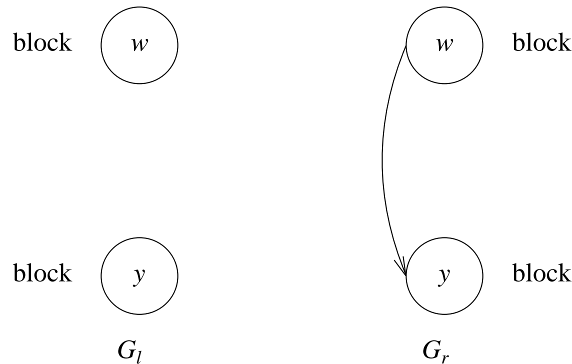

Figure 14

The predicate for this rule is **true** if and only if there is no directed path from the node y to the node w and y is not a successor of w.

Rule 2.5 Adding section omission links
A mechanism to bypass the section currently scanned has been provided by the links connecting successive *section head* nodes. The rule below allows several sections to be omitted in a single move through the graph without the need to go through the section headers. Again, to preserve acyclicity, only sections which are logically forward of the relevant section may be omitted.

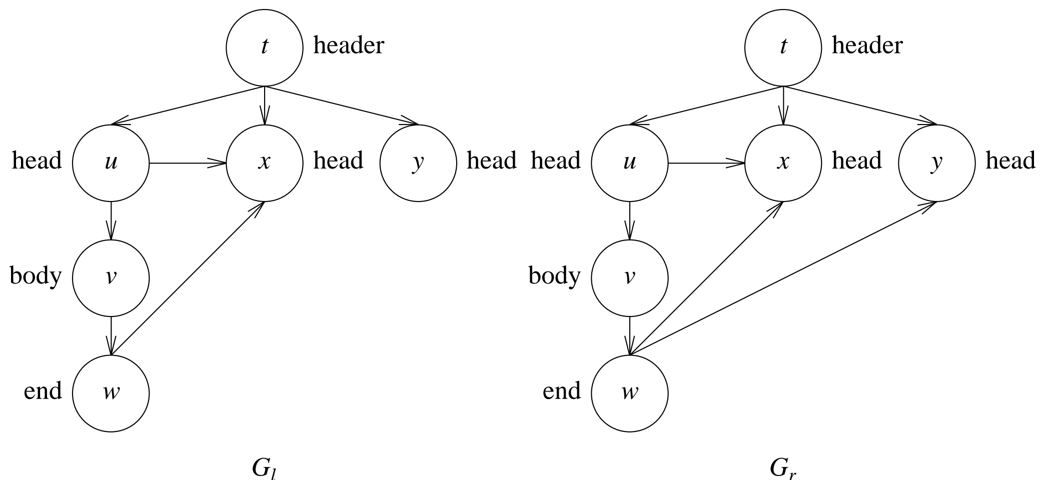

Figure 15

The predicate associated with this rule is **true** if and only if there is no directed path from the node y to the node w.

Theorem 5: The GMS described above is consistent.
Proof: Similarly we need only show that derived graphs are acyclic. Suppose the contrary. Let S denote the set of rules described; $H \in \Delta(S)$ be a directed graph containing a cycle; and $G \in \Delta(S)$ be a document graph such that $G \rightarrow H$. That G and H are well-defined is immediate from our initial assumption and the definitions of $\rightarrow$ and $\Delta(S)$. Consider the rule $R = < P, G_l, G_r > \in S$ such that $G \rightarrow H$ on applying R. By the choice of H, R cannot be a rule which deletes a link or an object from G since such rules cannot *create* cycles in the given system. Hence R is one of (2.1)–(2.6). It should be obvious that none of the rules (2.1), (2.2) or (2.3) can yield a cycle. Each of the rules (2.4)–(2.6) adds a directed link from some node w to some node y. This could only create a directed cycle in G if there were already a directed path from y to w, but if this is so then the associated predicate, P, of R prohibits application of the rule to G. It follows that no rule in R exists with the property needed hence our opening assumption is contradicted. It follows that S is consistent as claimed. □

Mathematical papers

The presentation of results in mathematical papers provides one of the best examples of a highly structured logical organization: a framework of technical definitions is set up; supporting lemmata are stated and proved, these being arranged in a precise sequence so that the final lemma may require an appreciation of those preceding it; then a series of theorems are stated and proved. In addition each theorem may spawn a related collection of corollaries. The stylistic discipline imposed by these universally accepted conventions is powerful enough to support structures which, in their purest form, sustain a mathematical argument without the intrusion of natural language (see e.g. Reference [18]).

In developing mathematical papers with computer aid it is imperative that the underlying document representation, at the very least, does nothing which can lead to this form being undermined: ideally the computer representation should exploit the organization that will be present in the final paper to assist the writer and greatly simplify the coordination of multiple author activity. The cult of semantic anarchy beloved of hypertext and similar ‘methodologies’ indicates that these are singularly ill suited to cope with such demands. Although a number of coherent readings will be possible for papers of this nature — one may wish to concentrate on the development of a single theorem or want to avoid some, possibly all, proofs in the paper — any valid analysis will follow the progression corresponding to the order in which the sections of interest would be encountered in a complete reading. At all levels — from elementary school texts to the most sophisticated specialist expositions — the reader or author of mathematical work cannot hope to gain or produce anything of value unless the order imposed by this regime is strictly adhered to. Consequently those liberal document representation techniques which act against the concept of order will be worse than useless as tools with which to develop mathematical papers.

We recall that even though such systems might be constrained to act in a suitable manner such an approach does not really deal with the fundamental objection being made for the reasons discussed in the introduction above.

The rules below describe a graph modification system capable of manipulating mathematical documents. The global structure of the graphs reflects the organization of such papers described above while facilities are provided to allow a large number of different partial readings each of which is valid.

In the description below we again omit details of rules for deleting objects. In addition since the graph structures for handling theorems and corollaries are identical to those for organizing lemmata we describe only the latter in detail.

Rule 3.1: Creating a mathematical paper
Papers are viewed as containing a header; a body of definitions; followed by a collection of lemmata; followed by a sequence of theorems and corollaries.

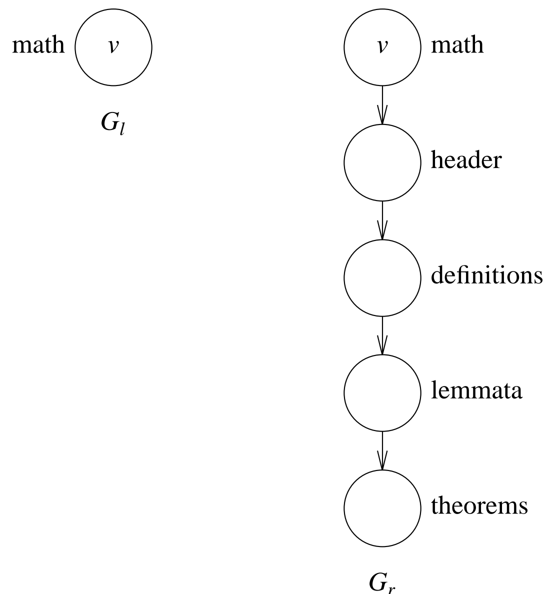

Figure 16

The predicate for this rule is the constant value **true**.

Rule 3.2: Expanding the definition section

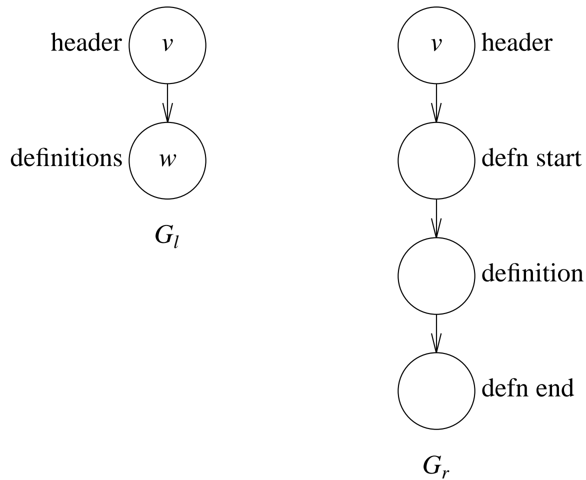

Figure 17

Again the predicate for this rule is the constant **true**.

Rule 3.3: Adding a new definition

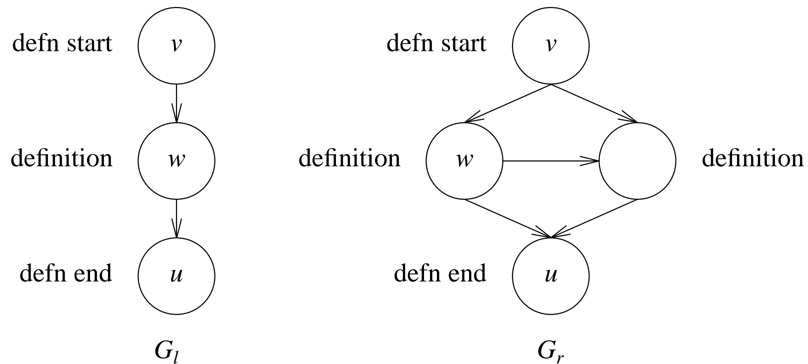

Figure 18

The predicate for this rule is **true** if and only if the node w has exactly one outgoing edge.

Rule 3.4: Expanding the lemma section

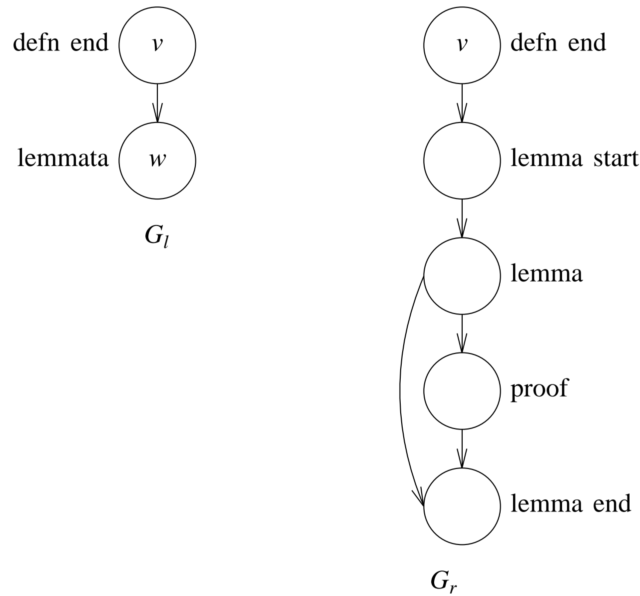

Figure 19

For this rule $P(G) =$ **true**.

Rule 3.5: Adding a lemma

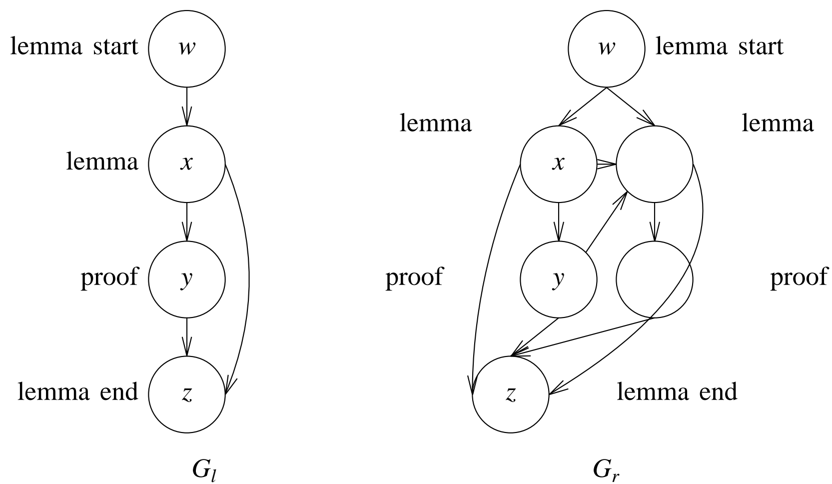

Figure 20

For this rule $P(G)$ holds if and only if the node w has exactly two outgoing edges.

A similar set of rules is used for expanding and developing the *theorems* section. The proof that the total system is consistent is similar to that of Theorem 5.

CONCLUSION

In this paper we take it as understood that any document management system must have a characterization of the documents it manages. We also claim that different classes of documents have different structures, and that a fruitful way of describing these structures is as a directed acyclic graph together with a consistent set of modification rules appropriate to that class. Other descriptions could be used; all of the documents discussed in this paper could be described as a finite string of ASCII characters, but this would lose the structure characteristic of the different types of documents, an understanding of which is required by both author and reader. If the document management system is to assist the creative process it must recognize the structural conventions and aid the author in their observance. An author using a system with the finite string of ASCII text model must impose these structures for himself: this is hard enough for a single author, but is likely to become impossible with a multi-author document.

One application of these ideas would therefore be an authoring tool to support the creation of documents of a given class by multiple authors. Enforcement of the modification rules would ensure that the structure appropriate to the document was maintained whatever modifications were made and irrespective of the author who made them.

In summary we believe that the model of documents put forward here, and exemplified by several typical examples, forms the theoretical basis of an efficient and extensible document management system, which would support the authorship, whether individual or collaborative, of a range of document types observing sensible structural conventions.

REFERENCES

1. R. Koo, ‘A model for electronic documents’, *ACM SIGOIS Bulletin*, **10** (1), 23–33 (1989).
2. R. Furuta, ‘An object-based taxonomy for abstract structure in document models’, Internal report, University of Maryland (1989).
3. P. D. Stotts and R. Furuta, ‘Adding browsing semantics to the hypertext model’, *Proc. ACM Conf. on Document Processing Systems*, pp. 43–50 (1988).
4. P. J. Brown, ‘Hypertext: the way forward’, in *Document Manipulation and Typography*, J. C. van Vliet ed., Cambridge University Press, Cambridge, pp. 183–191, (1988).
5. J. Conklin, ‘Hypertext: an introduction and survey’, *Computer*, **20** (9), 17–41 (1987).
6. G. D. Kimura and A. C. Shaw, ‘The structure of abstract document objects’, *Proc. ACM Conf. on Office Information Systems*, pp. 161–169 (1984).
7. N. Delisle and M. Schwartz, ‘Neptune: a hypertext system for CAD applications’, *Proc. ACM Intnl. Conf. on Management of Data*, pp. 132–143 (1986).
8. N. Meyrowitz, ‘Intermedia: the architecture and construction of an object-oriented hypermedia system and applications framework’, in *Proc. Object-Oriented Programming Systems Languages and Applications*, pp. 186–201, (1986).
9. *Standard ECMA-101 Office Document Architecture*, European Computer Manufacturers Association, (1985).
10. R. H. Thomas, H. C. Forsdick, T. R. Crowley, G. G. Robertson, R. W. Schaaf, R. S.

Tomlinson, and V. M. Travers, ‘Diamond: a multimedia message system built upon a distributed architecture’, *Computer*, **18** (12), 65–78 (1985).

11. S. Christodoulakis, M. Theodoridou, F. Ho, M. Papa, and A. Pathria, ‘Multimedia document presentation information extraction and document formation in minos: a model and a system’, *ACM Transaction on Office Information Systems*, **4** (4), 345–383 (1986).
12. E. Bertino, F. Rabitti, and S. Gibbs, ‘Query processing in a multimedia document system’, *ACM Transaction on Office Information Systems*, **6** (1), 1–41 (1988).
13. M. Nagl, *A Tutorial and Bibliographic Survey on Graph Grammars*, Springer-Verlag, New York, (1978). Lecture Notes in Computer Science.
14. J. Joyce, *Finnegans Wake*, Faber and Faber, 1939. Corrected impression 1975.
15. T. J. M. Bench-Capon, P. E. S. Dunne, and A. M. McEnery, ‘A document model to support cohesive ties in text’, Internal Report CS/CSCW/4/1989, University of Liverpool (1989).
16. J. Hopcroft and J. D. Ullman, *Introduction to Automata Theory Languages and Computation*, Addison-Wesley, Reading, MA, (1979).
17. A. M. Turing, ‘On computable numbers with an application to the *Entscheidungsproblem*’, *Proc. London Math. Soc.*, **2** (42), 230–265 (1936).
18. B. Russell and A. Whitehead, *Principia Mathematica*, Cambridge University Press, .