Proceedings of the 21st IASTED International Conference
APPLIED INFORMATICS
February 10-13, 2003, Innsbruck, Austria

A DTD COMPLEXITY METRIC

Ron McFadyen Yangjun Chen*
Department of Business Computing, University of Winnipeg
515 Portage Avenue, Winnipeg
Canada R3B 2E9
{r.mcfadyen, ychen2}@uwinnipeg.ca

* Supported by NSERC 239074-01 (242523) (Natural Sciences and Engineering Council of Canada)

ABSTRACT

In this paper, we propose a metric for measuring the complexity of a DTD. For the same purpose, different designers may create different DTDs. A complexity metric can assist us to evaluate and compare their different merits. Through the recognition of strongly connected components, the metric accommodates the recursive relationship of elements in a DTD. The metric presented here can be expressed in a simple and informal way involving the number of elements, connectors, appearance indicators and back edges.

KEY WORDS

cyclomatic, complexity, DTD, metric, XML

1. Introduction

Recently, complexity of document structures has received significant attention. Complexity is the degree to which a system or component has a design or implementation that is difficult to understand and verify [1]. In the literature [2, 3] there are references to the complexity of DTDs, but there is no specification of how that complexity is measured.

Many approaches to complexity have been proposed. For example, in [4] the authors considered geometric complexity of documents scanned into a computing system. [5] uses a metric called *inverse document frequency* that counts words to measure document complexity. [6] is concerned with cognitive complexity, and proposes measuring text comprehension complexity by focusing on unfamiliar word usage. [7] discusses the complexity of hypertext documents, and draws a parallel between the development and maintenance of software and the development and maintenance of hypertext documents.

In this paper, we are concerned with the concept of document complexity as it applies to the structure defined in a *Document Type Description* (DTD). Both the structure and its complexity measurement are defined.

To measure a DTD, we map it into a directed graph. We use this graph as a basis for illustrating and measuring DTD complexity. Our approach adapts the concept of cyclomatic complexity to the structure of DTDs.

The paper is organised as follows. In Section 2, we review DTDs, simplified DTDs, and DTD graphs in some detail. In section 3, the concept of cyclomatic complexity is described, which was first introduced in [8] to measure software complexity. Section 4 discusses DTD complexity, and Section 5 is devoted to a DTD complexity metric. Finally, a short conclusion is set forth in Section 6.

2. Document Type Description

In a Document Type Description (DTD), we specify the components and structure of an XML document. For a simple illustration, see a possible DTD for letter documents shown in Figure 1.

```
1  <!DOCTYPE  letter [
2  <!ELEMENT  letter (head, address, greeting, body, closing, sig)>
3  <!ATTLIST   letter
4                  filecode NUMBER #REQUIRED
5                  secret (yes | no) “no”>
6  <!ELEMENT  body (para+)>
7  <!ELEMENT  head (to, from, date)>
8  <!ELEMENT  to (person)>
9  <!ELEMENT  from (person)>
10 <!ELEMENT  person (firstname?, lastname, address)>
11 <!ELEMENT  (date, firstname, lastname, address, greeting,
12                 closing, sig)   (PCDATA)>
13 <!ELEMENT  para ((text | emph)*, sub_para*)>
14 <!ELEMENT  emph (text | sub_para)>
15 <!ELEMENT  sub_para (para)>
16 <!ELEMENT  text (#PCDATA)>
17 <!ATTLIST   text italic (yes | no) “yes”>
18 <!ENTITY    salute “Dear”>
19 ]>
```

Figure 1. An XML DTD

An XML document is defined as having elements and attributes [9, 10]. Elements are always marked up with tags; and an element may be associated with several attributes to identify domain-specific information. For example, element *letter* has two attributes: *filecode* and *secret* (see lines 3, 4, 5 of Figure 1). In addition, a special attribute, '*id*', may be specified once for each element. The '*id*' attribute uniquely identifies an element within a document and can be referenced through an 'IDREF' field in another element. The 'IDREF' is untyped. An element may have sub-elements and sub-element structure is specified using the operators '*' (set with zero or more elements), '+' (set with one or more elements), '?' (optional), '|' (or) and ',' (and). Further, we distinguish between primitive and complex elements. A primitive element contains only data of primitive types such as integer, string and '#PCDATA' (which is more or less comparable to string) while a complex element contains one or more sub-elements which are primitive or complex by themselves. For example, elements *head*, *address*, *greeting*, *closing* and *sig* are all primitive (see lines 11, 12). But element *letter* is a complex element (see line 2). It contains six sub-elements, of which both *body* and *head* are themselves complex (see lines 6 and 7). In addition, attention should be paid to the line starting with '<!ENTITY' which introduces "replacement text" (see line 18). That is, if a string of the form '&salute;' appears in any concrete document conforming to the DTD, this string will be substituted with "Dear" (see [10] for XML entity definition.)

2.1 DTD Simplification

DTDs tend to be complicated. For example, we could specify an element type as <!ELEMENT a ((b|c|e)?, (e?|(f?, (b, b)*))*)>, where 'a' is an element being specified; 'b', 'c', 'e' and 'f' represent other element types. Such a structure is heavily nested and cannot be represented by a (labeled) directed graph. For this reason, we introduce *virtual element types* to simplify element definitions appearing in a DTD; but without damaging the semantics of a DTD as discussed in [11]. For instance, for <!ELEMENT a ((b|c|e)?, (e?|(f?, (b, b)*))*)>, six virtual element types, defined according to *production rules*, are introduced:

A ← b|c|e
B ← b, b
C ← f?
D ← e?
E ← C, B*
F ← D|E

Now, using these virtual types, we can specify the above element type as

<!ELEMENT a (A?, F*)>.

Each production rule can be considered an element type definition, but *virtual* since it is not present in the original DTD. In fact, the original DTD is equivalent to that shown in Figure 2, in which six virtual elements are introduced.

<!ELEMENT a (A?, F*)>
<!ELEMENT A (b|c|e)>
<!ELEMENT B (b, b)>
<!ELEMENT C (f?)>
<!ELEMENT D (e?)>
<!ELEMENT E (C, B*)>
<!ELEMENT F (D|E)>

Figure 2. Rewriting with simple elements

In this way, the nested structure in an element specification is removed. In [11], an element specification without 'nesting' is called a *simple element specification*.

Definition 1. A simple element specification is of the form: <!ELEMENT α ($\alpha_1\beta$ ... $\alpha_{n-1}\beta\alpha_n$)>, where α is an element type, possibly virtual, each α_i represents a single element type (possibly virtual and possibly decorated with '?', '+' or '*') and $\beta \in$ {',', '|'}.

We categorize simple element specifications one step further.

Definition 2. In a simple element specification <!ELEMENT α ($\alpha_1\beta$ ... $\alpha_{n-1}\beta\alpha_n$)>, if β = ',', then it is an *And specification*; if β = '|', then it is an *Or specification.*

If a DTD contains only simple element specifications, it is called a *simple DTD*.

Obviously, the goal of DTD simplification is to transform an arbitrary DTD into a simple DTD augmented with virtual elements. From the above discussion, we see that this can be done straightforwardly without any difficulty. First, we introduce a series of virtual element types that can be established by analysing each element specification of the original DTD individually. Then, we use the following transformations:

α_1** –> α_1*,
α_1*? –> α_1*,
α_1?* –> α_1*,
α_1?? –> α_1?,

to reduce many unary operators to a single unary operator as discussed in [2].

Example 1. A simplified DTD for <!ELEMENT a ((b|c|e)?, (e?|(f?, (b, b)*))*)> is shown in Figure 2.

Example 2. The DTD shown in Figure 1 can be transformed into a simplified DTD as shown in Figure 3. By this DTD simplification, only one virtual element: <!ELEMENT A (text | emph)> is generated (see line 13.1) and the element specification <!ELEMENT para ((text | emph)*, sub_para*)> is changed to <!ELEMENT

para (A*, sub_para*)> (see line 13), where 'A' represents a virtual element type.

```
1  <!DOCTYPE  letter [
2  <!ELEMENT  letter (head, address, greeting, body, closing, sig)>
3  <!ATTLIST   letter
4                  filecode NUMBER #REQUIRED
5                  secret (yes | no) "no">
6   <!ELEMENT      body (para+)>
7   <!ELEMENT      head (to, from, date)>
8   <!ELEMENT      to (person)>
9   <!ELEMENT      from (person)>
10 <!ELEMENT      person (firstname?, lastname, address)>
11 <!ELEMENT      (date, firstname, lastname, address, greeting,
12                  closing, sig)    (PCDATA)>
13  <!ELEMENT      para (A*, sub_para*)>
13.1 <!ELEMENT    A (text | emph)>
14   <!ELEMENT    emph (text | sub_para)>
15   <!ELEMENT    sub_para (para)>
16   <!ELEMENT    text (#PCDATA)>
17   <!ATTLIST    text italic (yes | no) "yes">
18   <!ENTITY     salute "Dear">
19   ]>
```

Figure 3. DTD of Figure 1 rewritten with simple elements.

2.2 DAO Graph

A *Decorated DTD And/Or graph* (called a DAO graph) can be established for any simplified DTD, and is defined as follows.

Definition 3. A DAO graph for a set *S* of simple element specifications is a directed graph, where there is a node for each (decorated) element type in *S* and an edge from 'a' to 'b' if there exists a simple element type specification of the form: <!ELEMENT a (... b ...)> in *S*. If 'b' is of the form: α?, then the edge is labelled with '?'. If 'b' is of the form: α*, then the edge is labelled with '*'. If 'b' is of the form: α+, then the edge is labelled with '+'. Each node of the graph is characterized as an *And-node* or an *Or-node*: if an element *n* is an And specification, then the node *n* is an And node, otherwise *n* is an Or node.

Example 3. For the transformed DTD shown in Figure 3, we can draw a DAO graph as shown in Figure 4.

Note that a DAO graph also represents a process one can follow to generate a document, or to process a document conforming to the DTD. During a traversal of the graph, one could generate an XML document conforming to the DTD.

As with a normal directed graph, we distinguish among four kinds of edges in a DAO graph, which represent different semantic relationships of elements and imply different complexities. To do this, we will assign an integer to each node encountered during a depth-first traversal of a DAO graph, which is used to number the nodes in the order they are reached during the traversal. The number assigned to a node *v* is denoted *dfsnumber*(*v*). This numbering can be used to distinguish among tree edges, forward edges, back edges and cross edges. They are defined as follows.

(i) *tree edge*: An edge *e*: $v \rightarrow u$ is a tree edge if *u* is reached from *v* when it is scanned and *u* has not been visited before (then at this moment *dfsnumber*(*u*) = 0 if we initially assign each *dfsnumber* with 0.)

(ii) *forward edge*: An edge *e*: $v \rightarrow u$ is a forward edge if when it is scanned for the first time *dfsnumber*(*u*) > *dfsnumber*(*v*) > 0.

(iii) *back edge*: An edge *e*: $v \rightarrow u$ is a back edge if when it is scanned for the first time, *dfsnumber*(*v*) > *dfsnumber*(*u*) > 0 and at the same time *u* is an "ancestor" of *v*. (We say a node *x* is an ancestor of node *y* if they appear in the same path and *x* is visited before *y* during the depth-first traversal.)

(iv) *cross edge*: An edge *e*: $v \rightarrow u$ is a cross edge if when it is scanned for the first time, *dfsnumber*(*v*) > *dfsnumber*(*u*) > 0 but *u* is not an "ancestor" of *v*.

See [11] for a detailed description.

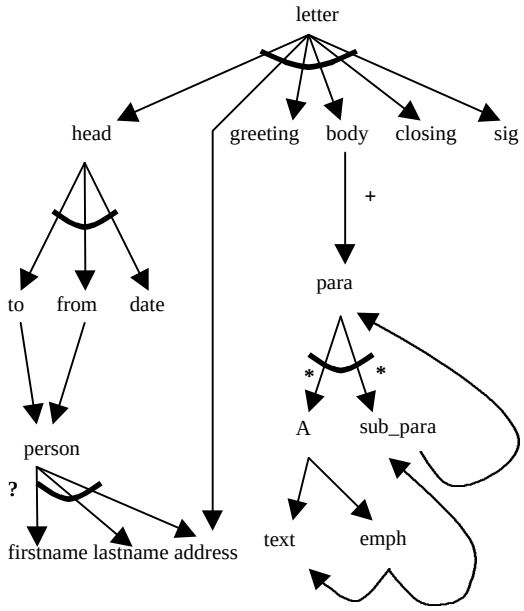


Figure 4. A DAO graph for the letter DTD.

3. Cyclomatic Complexity

Cyclomatic complexity measures the number of linearly-independent paths through a program module and was introduced by Thomas McCabe in 1976 [8]. Cyclomatic complexity provides a single ordinal number that can be compared to the complexity of other programs. A low

cyclomatic complexity contributes to a program's understandability and indicates it is amenable to modification at lower risk than a more complex program.

Cyclomatic complexity has also been extended to encompass *design complexity* [12] and *data complexity* [13]. Design complexity measures the amount of interaction between decision logic and subroutine calls. Data complexity measures the amount of interaction between decision logic and data references.

Given a program, a flowgraph can always be associated with it. In the graph, each node corresponds to a block of code where the flow is sequential, and arcs correspond to branches in the program. The cyclomatic complexity, *CC*, of a graph *G* with *n* vertices and *e* edges is:

$CC(G) = e - n + 2$

Cyclomatic complexity is easily determined using the number of decision statements in a program, and so $CC(G)$ can be stated as:

$CC(G)$ = number of decision statements + 1.

For instance, for the program shown in Figure 5(a), the cyclomatic complexity is $CC(G) = e - n + 2 = 8 - 7 + 2 = 3$ (see Figure 5(b)); note there are two decision statements in the program.

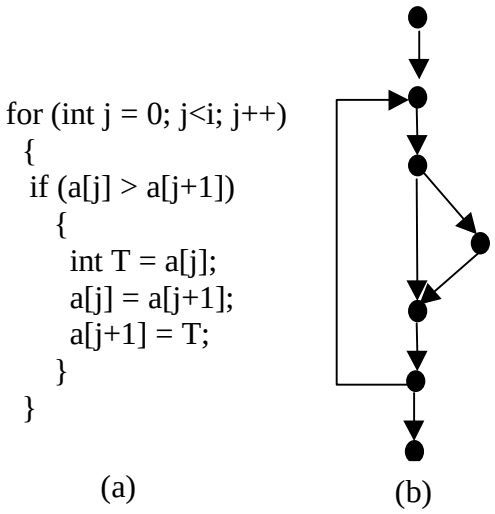


Figure 5. Program and corresponding flowgraph

[12] applied cyclomatic complexity to the structure chart (hierarchy tree) used in Structured Design. Design complexity that does not affect procedure calls is removed and analysis is done on a reduced graph. Design complexity measures the amount of interaction between decision logic and subroutine calls. For example, Figure 6 illustrates a design tree of 3 modules where module 1 conditionally calls modules 2 and 3.

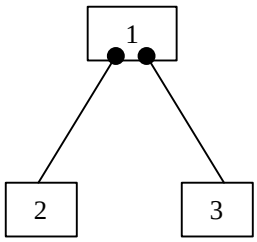


Figure 6. Module 1 conditionally calls modules 2 and 3.

Each module, G_i, has an individual design complexity, $iv(G_i)$. The design complexity of a module *M*, S_o, is defined as

$$S_o = \sum_{i \in D} iv(G_i)$$

where *D* is the set of descendants of *M* unioned with *M*.

4. DTD Complexities

DTDs are declarative descriptions for the contents of a document. As discussed above, we can represent a simplified DTD with a DAO graph. In this section, we use the DAO graph to present the complexities of DTDs, and relate each of these complexities to similar procedural representations where cyclomatic complexity has been applied. A DAO graph comprises the following:

Nodes:
- And nodes
- Or nodes

Decorations

Edges:
- Tree edges
- Forward edges
- Cross edges
- Back edges

Next, we discuss node types and decorations in order to obtain node complexities for each node in a DAO graph. Then, we consider edges and present a reduced DAO graph in order to eliminate cycles if any.

4.1 Nodes

According to the node type, we associate an initial node complexity (*NC*) for each node in the graph.

4.1.1 And Nodes

An And-node represents a situation where the DTD user has no choice. When creating a document according to an element specification that is an And-node, the user must include all sub-elements. All documents formed in this way will have the same structure; there is no variability.

A process that a user can follow is represented in a flowgraph as shown in Figure 7. Since the process in Figure 7(b) has a cyclomatic complexity of 1, we assign an initial node complexity, *NC*, of an And-node to be 1 as well.

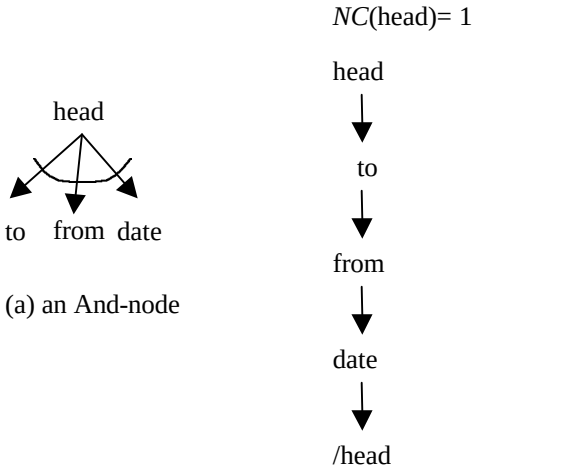


Figure 7. And-node complexity

4.1.2 Or Nodes

An Or-node represents a situation where the DTD user has a choice to make. When creating a document according to the DTD, the user must choose to include one of the specified elements. Documents created according to the DTD, can vary from one another; the DTD allows for variability when an Or-node is encountered. The process followed can be represented in a flowgraph as shown in Figure 8. The process in Figure 8(b) has a cyclomatic complexity of 2, and so we assign 2 as the node complexity associated with this Or-node. In general, the initial node complexity of an Or-node is the number of alternative choices.

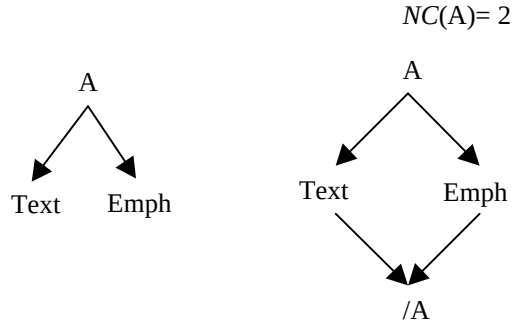


Figure 8. Or-node complexity

Figure 9 illustrates the initial complexities assigned to nodes in terms of the assignment strategy discussed above.

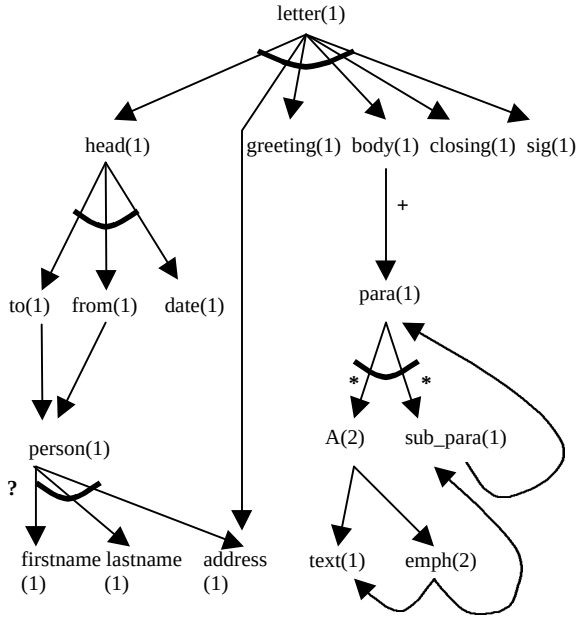


Figure 9. Initial node complexities

4.2 Decorations

There are three types of decorations that appear in a DAO graph: '*', '?', '+'. These decorations impose restrictions on how many times a subtree can be utilized, and hence increase the complexity of the DTD. For each decoration, the user must choose the number of times the subtree will be utilized. The process followed can be represented in a flowgraph as shown in Figure 10. For each of '*', '?' and '+', the node complexity of And and Or nodes increase by 1. Figure 11 illustrates node complexities adjusted for decorations.

4.3 Edges

Edges can be of four types: tree, forward, cross, and back, which implies different complexities. As discussed in 2.2, the different edge types can be distinguished based on depth-first numbering.

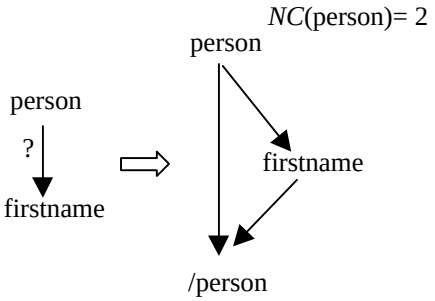
Tree Edges.
Tree edges represent the normal hierarchical structure of a document, and as such, do not increase the complexity of a DTD. If a DAO graph contains only Tree edges, then the DTD is a special case; its just a simple hierarchy.

Forward and Cross Edges.
Forward and Cross edges indicate that some aspects of the DTD are being reused. These simplify DTD definitions, and are not considered to increase the complexity of a DTD. If a DTD contains only Tree edges, a DTD designer may recognize similar patterns appearing in different

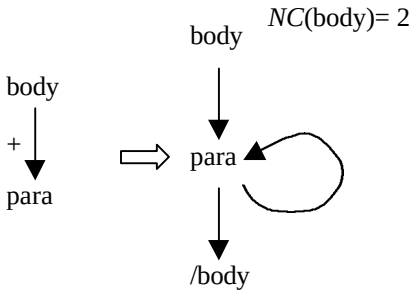
subtrees and the designer may decide to generalize these patterns and reuse element definitions. Such generalization is normally considered to simplify the DTD. The DAO graph for the letter DTD (Figure 4) illustrates three reuse situations: person, address, and text. Hence, we ignore Forward and Cross edges in complexity calculations.

Back Edges.

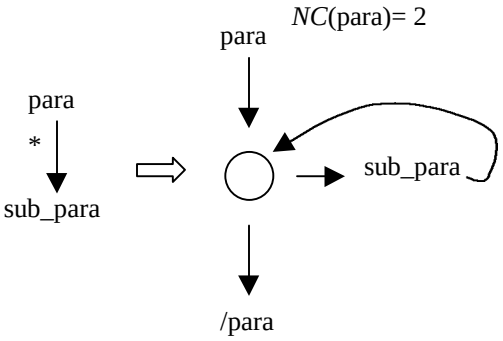
Back edges represent a special case of reuse; they are labelled as back edges because, during a tree traversal, they allow for a prior node to be encountered again. This is commonly referred to as recursion, and is a powerful DTD design feature. When a similar process structure is analyzed for cyclomatic complexity, the added presence of a back edge increases complexity by one, see Figure 12.



(a) equivalent flowgraph for '?'



(b) equivalent flowgraph for '+'



(c) equivalent flowgraph for '*'

Figure 10. Decorations represented procedurally

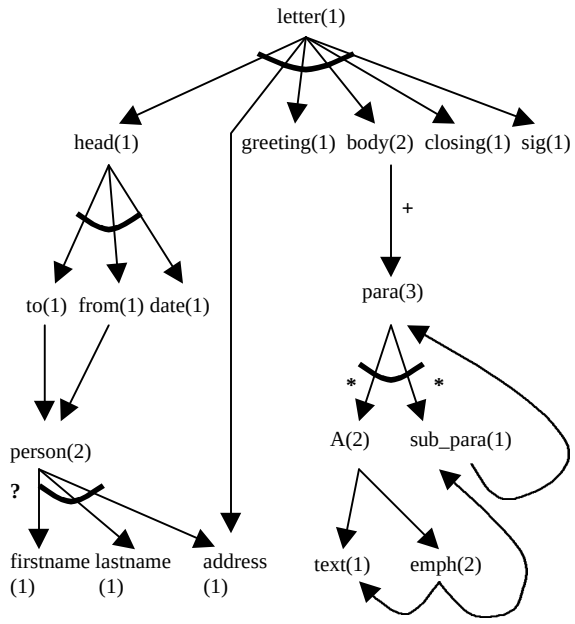
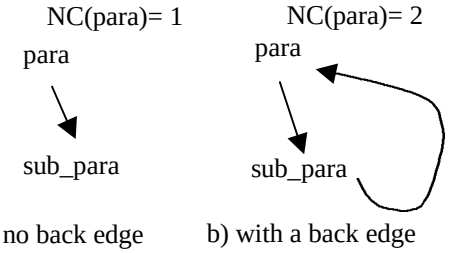


Figure 11. Adjusted node complexities



a) no back edge b) with a back edge

Figure 12. Back edges increase complexity

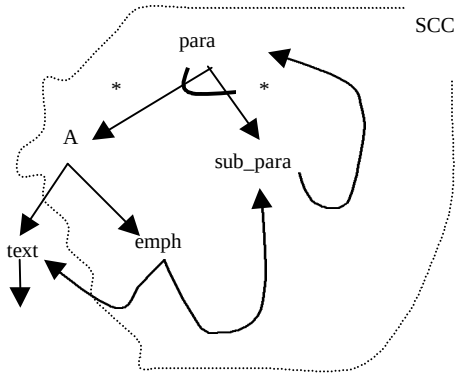
[11] discusses *strongly connected components* (SCC) in a DTD specification. An SCC can be complex, as shown in Figure 13. The SCC in Figure 13 has an And-node, Or-node, decoration '*', and two back edges. Since every node in an SCC is involved in a cycle, we reduce the SCC to a single virtual node with a node complexity, *NC*(SCC) calculated as

$$\sum_{c \in D} NC(c) + b$$

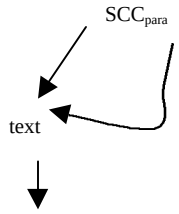
where D is the set of nodes in the SCC and b is the number of back edges in the SCC. For our example, the $NC(SCC_{para})$ is

$$\begin{aligned} NC(SCC_{para}) &= NC(para) + NC(\text{A}) + NC(\text{emph}) + \\ &\quad NC(\text{sub\_para}) + 2 \\ &= 3 + 2 + 2 + 1 + 2 \\ &= 10. \end{aligned}$$

Replacing SCCs with virtual nodes results in a reduced DAO graph without cycles.



(a) SCC rooted at para



(b) SCC replaced by a virtual node

Figure 13. A complex SCC rooted at para.

5. A DTD Complexity Metric

We have discussed the internal complexities of an individual DTD component C. In this section, we present a complexity metric, *DC*, for a DTD and for DTD elements. We define the DTD complexity, DC, of an element *E* of a DTD as

$$DC(E) = \sum_{c \in D} NC(c)$$

where *D* is the set of all elements accessible from *E*, including *E*, in the reduced DAO graph. Note that in our example, SCC_{para} replaces the elements of the SCC. The complexity of a DTD is just *DC*(*R*) where *R* is the root of the DTD.

Consider the element person in Figure 4. The DTD complexity of person is

DC(person) = *NC* (person) + *NC* (firstname)
+ *NC* (lastname) + *NC* (address)
= 2 + 1 + 1 + 1
= 5

Due to cross and forward edges, DTD complexity is not upwardly additive. For instance, *DC*(head) *DC*(to) + *DC*(from) + *DC*(date). This is because both *DC*(to) and *DC*(from) include *NC* (person).

The DTD complexity of the letter DTD is

DC(letter)
= *NC*(letter) +*NC*(head) +*NC*(address)
+*NC*(greeting) +*NC*(body) +*NC*(closing) +*NC*(sig)
+*NC*(to) +*NC*(from) +*NC*(date) +*NC*(SCC_{para})
+*NC*(person) +*NC*(text) +*NC*(firstname)
+*NC*(lastname)
= 1+1+1+1+2+1+1+1+1+1+10+2+1+1+1
= 26

Values of DC for the reduced DAO graph are shown in Figure 14. Next to each node is a pair of values: (*NC*, *DC*).

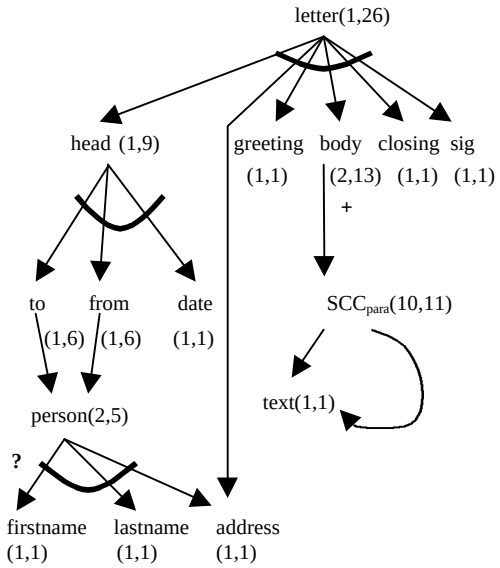


Figure 14. Complexity metric values

There are a number of properties of the complexity metric *DC*:

1. If there are no common elements (no forward, cross, or back edges in the DAO graph), then *DC* is upwardly additive.
2. The elements that are leaf vertices in the DAO graph have complexity of 1.
3. Where n is the number of elements in the design, *DC* is bounded as:

$$n \le DC \le \sum_{i=1}^{n} NC(G_i)$$

4. Adding an element to a design increases DC by at least 1.
5. Simplifying a DTD through generalization and reuse of an element *M* reduces *DC* by *DC*(*M*).
6. *DC* is defined for any element *T* as

$$DC(T) = \sum_{i} NC(G_i)$$

6. Conclusion

We have presented a metric for evaluating the complexity of DTDs. With reference to the elements in the DAO graph, this metric, *DC*, can be expressed in a simple and informal way as:

> the number of elements, plus the number of '|'s in element definitions, plus the number of decorations, plus the number of back edges.

The metric can be easily modified to generate other metrics that give different weightings to different aspects of DTD definitions as follows.

$$DC(dtd) = W_{element} * No_{elements} + W_{alternative} * No_{|'s} + W_{decoration} * No_{decorations} + W_{recursion} * No_{back\ edges}$$

Future empirical research can provide information regarding average and variance in *DC* values, the numbers of elements, alternatives, decorations, and the numbers of the different edge types appearing in DTDs. This information can assist in comprehending the complexities present in DTDs and DTD libraries.

References

[1] IEEE standard glossary of software engineering terminology, *IEEE Standard 610.12-1990.*

[2] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, J. Naughton, Relational databases for querying XML documents: limitations and opportunities, *Proc. VLDB*, Edinburgh, Scotland, 1999.

[3] Chuang-Hue Moh, Ee-Peng Lim, Wee-Keong Ng, DTD-miner: a tool for mining DTD from XML documents, *Proceedings of the 2nd International Workshop on Advanced Issues of E-Commerce and Web-based Information Systems*, San Jose, CA. WECWIS 2000.

[4] Y. Y. Tang, Hong Ma, Xiaogang Mao, Dan Liu, C.Y. Suen, A new approach to document analysis based on modified fractal signature, *Third International Conference on Document Analysis and Recognition*, August 1995, 567-570.

[5] Ian Ruthven, Mounia Lalmas, Keith van Rijsbergen, Combining and selecting characteristics of information use, *Journal of the American Society for Information Science and Technology (JASIST), 53*(5), 2002.

[6] T. Klemola, A cognitive model for complexity metrics, *4th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, June 13, 2000.

[7] Pearl Brereton, David Budgen, Geoff Hamilton, Hypertext: the next maintenance mountain, *Computer*, December 1998, 49-55.

[8] T. J. McCabe, A complexity measure, *IEEE Transactions Software Engineering, 4*(2), 1976, 308-320.

[9] J. Clark, Comparison of SGML and XML, *http://www.w3.org/TR/NOTE-sgml-xml-971215*, December 1997.

[10] S.J. DeRose and D.D. Durand, *Making hypermedia work: A user's guide to HyTime* (London, Kluwer Academic Publishers, 1994).

[11] Yangjun Chen, Ron McFadyen, Fung-Yee Chan, Mapping DTDs to object-oriented schemas, *The 2nd International Conference on Web Information Systems Engineering (WISE2001)*, Kyoto, Japan, December 2001.

[12] T. J. McCabe and C. W. Butler, Design complexity measurement and testing, *Communications of the ACM, 32*(12), 1989, 1415-1425.

[13] D. Card, R. Glass, *Measuring software design quality* (Prentice Hall, 1990).