

# Specifying the Structural Properties of Software Documents\*

Jun Han

School of Computing and Information Technology

Monash University, McMahons Road

Frankston, Vic 3199, Australia

e-mail: jhan@monash.edu.au

phone: +61 3 9044604, fax: +61 3 9044124

---

\*A paper presented at the *6th International Conference on Computing and Information* (Peterborough, ON, Canada, May 1994), and appeared in the *Journal of Computing and Information*, 1:1333-1351, 1994.

# Specifying the Structural Properties of Software Documents

Jun Han

School of Computing and Information Technology  
Monash University, McMahons Road  
Frankston, Vic 3199, Australia  
e-mail: jhan@monash.edu.au

**Abstract.** One of the central issues involved in developing a software development environment is the consistency support for software documents and their relationships. To provide such support, the environment has to have the knowledge of the consistency requirements of the software documents concerned. In this paper, we introduce an approach to specifying the structural properties of software documents in a declarative style. Such a specification can be used to augment a generic tool to provide specialised structural consistency support for the software documents concerned. In particular, we show how composed relationships can be used to specify structural properties.

**Keywords.** Inter- and intra-document relationships, software consistency, software development environments and tools, software documents, structural properties.

## 1 Introduction

Software development environments are supposed to support the complete software development process and maintain all the documents produced in this process. Corresponding to the different phases of the software process, there are different kinds of software documents, such as the requirements, the specifications, the designs, the implementations, the testing suites, the maintenance records, and the technical documentations. These software documents are logically related to each other according to the software process. In principle, the software development environments should provide unified support for the software process and the software documents.

There have been diverse views on how to deal with the relationships between the software process and the software documents in a software development environment. In particular, they have led to the development of process-centred and product-centred software development environments. To provide unified support for the software process and the software documents, we follow a document-based approach and regard software development as a document-based process [10]. We assume that the end-product of a software development process is a *necessary history* of the development, rather than just the compilable or executable programs that result. This history includes not only the software documents involved in the software process, but also the relationships between these documents and the relationships between their components. These relationships characterise the traceability between and within software documents, which is necessary to facilitate process support.

One of the central issues involved in developing a software development environment is the consistency support for software documents and their relationships. To provide such support, one has to define the consistency requirements of the software documents and their relationships and develop the relevant computer-based tools which form part of the support environment. For example, programs written in a

particular language have consistency requirements with respect to that language's syntax and semantics, and tools supporting the development of these programs are developed according to the given syntax and semantics.

There have been different approaches to developing computer-based tools to support the development of software (or preparation of software documents). In particular, a large number of these approaches focus on the development of generic tools. A generic tool can be augmented by a description of the knowledge specific to a particular software development project or methodology, to obtain a project- or methodology-specific tool. For example, these approaches have been widely used and proven effective in developing language-based editors, such as in the Synthesizer Generator [6, 7]. In general, the generic approaches can be used to handle the syntax and/or static semantics of the software documents.

In this paper, we introduce a new approach for describing the structural properties of the software documents and their relationships where the structural properties, to a large extent, capture the static semantics of the documents and relationships concerned. The description can be used to augment a generic tool to provide specialised support for these documents and relationships. This approach allows the description of the structural properties in a declarative style as the Extended Backus-Naur Form (EBNF) does for the syntax of documents. This is in clear contrast to the operational style often found in existing approaches (e.g., attribute calculation). With our approach, we hope to achieve uniformity in style and effectiveness when describing the syntactic and structural aspects of software documents.

In section 2 of this paper, we present a brief overview of how the syntactic aspect of software documents and their relationships are dealt with in our approach. In section 3 we illustrate how the structural properties of the documents and relationships concerned are specified. In section 4 we review related work.

## 2 Software documents and their relationships

The development of a software system involves a range of software documents. There exist relationships between these documents and between their components. For instance, an abstract function in a design may be *implemented by* a concrete function in the implementation. As argued earlier, both the individual documents and their relationships should be captured in the support environment to facilitate traceability of the software development process. In this section, we present a brief overview of how the syntactic aspect of individual documents and their relationships is represented and described (a detailed discussion can be found in [4]).

### 2.1 Syntactic structures

A software document usually has a number of components, these components may have further lower-level components, and so on. This hierarchical structure is referred to as the *syntactic structure* of the software document. The software document and all its components at different levels are referred to as *document segments*.

For effective processing, the internal representation of documents in traditional language-processing tools, such as compilers and language-based editors, takes a form of syntax trees. In most cases, the notation for describing the syntactic structures of documents is a variation of the EBNF. For example, both the Synthesizer Generators [6, 7] and the UQ editors [11, 1, 9] adopt such a representation and description approach. In our approach, we also use syntax trees to represent software documents and use an augmented EBNF notation to describe their syntactic structures. To allow natural representation and description of software documents, our approach has two distinctive features. First, multiple documents can be represented and described at the same time. Second, the components of certain document segments can be unordered wherever enforcing a linear order is unnatural or inconvenient. Therefore,

a document segment may be *sequential* or *collective*. In general, a document may contain sequential and collective segments at various levels.

To illustrate the syntactic representation and description of individual documents in our approach, consider a small but typical software development example. Suppose that we are required to develop a program to calculate the Greatest Common Divisors (GCDs) of a file of positive integer pairs. The development of such a program may involve: *a requirements document* in natural language; *a specification document* in a formal specification language Z [8]; *a design document* also in Z; *an implementation document* in Pascal. Due to space limitation, we only consider the design and implementation documents. The following is the design document:

*We isolate the calculation of the GCD of one pair of positive integers as a sub-task, which can be specified as a function  $D\_GCD$  in Z:*

$$\begin{array}{|l}
 \hline
 D\_GCD : (\mathbb{N}_1 \times \mathbb{N}_1) \rightarrow \mathbb{N}_1 \\
 \hline
 \forall a, b : \mathbb{N}_1 \times \mathbb{N}_1 \bullet \\
 \quad D\_GCD(a, b) \text{ div } a \\
 \quad D\_GCD(a, b) \text{ div } b \\
 \quad \neg \exists z : \mathbb{N} \bullet z > D\_GCD(a, b) \wedge z \text{ div } a \wedge z \text{ div } b \\
 \hline
 \end{array}$$

*Using this function, the entire task can be specified as a Z schema  $D\_GCDs$ , whose primary concern is the sequential input/output relationship required:*

$$\begin{array}{|l}
 \hline
 D\_GCDs \\
 \hline
 i? : \text{seq}(\mathbb{N}_1 \times \mathbb{N}_1) \\
 o! : \text{seq}((\mathbb{N}_1 \times \mathbb{N}_1) \times \mathbb{N}_1) \\
 \hline
 i? = \text{first} \circ o! \\
 \text{second } o! = D\_GCD \circ (\text{first} \circ o!) \\
 \hline
 \end{array}$$

The implementation document is as follows:

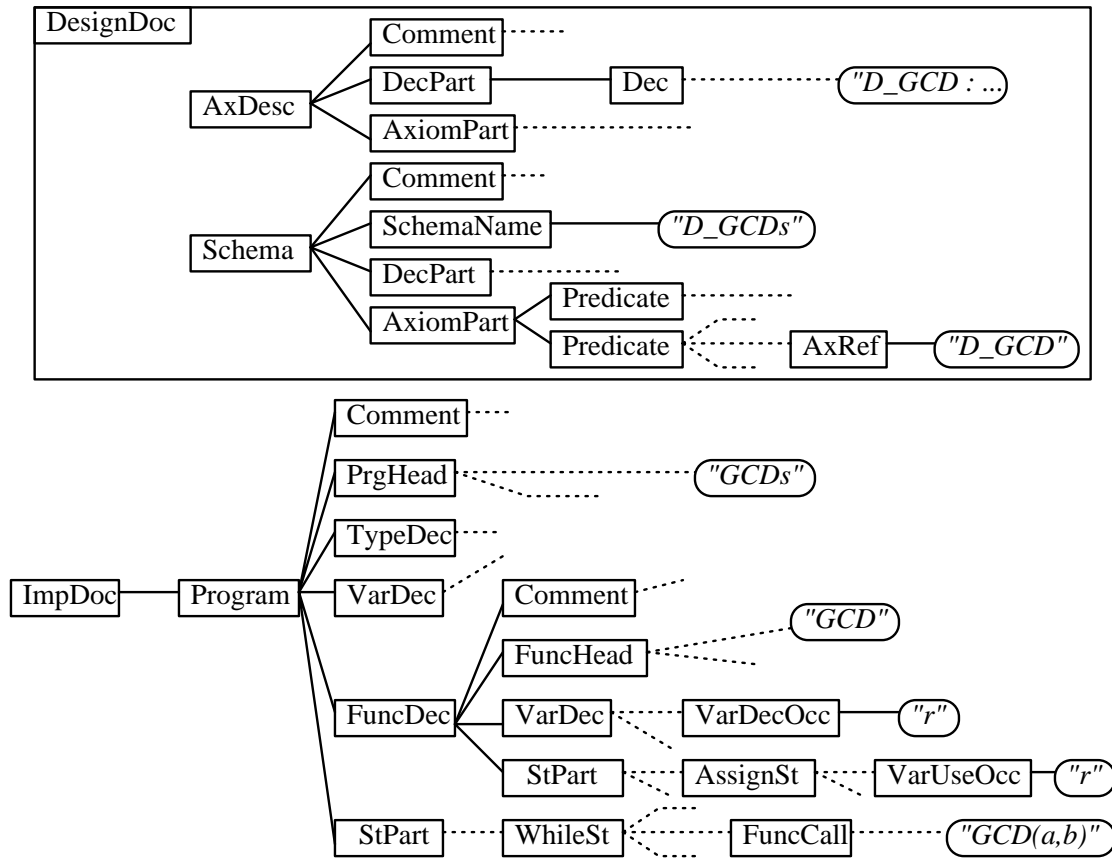
*The Z function D\_GCD can be implemented as a Pascal function using Euclid's algorithm:*

```
function GCD(a, b : positiveInteger) : positiveInteger;  
  var r : positiveInteger;  
  begin  
    r := a mod b;  
    while r <> 0 do begin a := b; b := r; r := a mod b end;  
    GCD := b  
  end;
```

*The entire task can be implemented as a Pascal program:*

```
program GCDs(input, output);  
  type positiveInteger : 1 .. maxint;  
  var a, b : positiveInteger;  
  function GCD ... ;  
  begin  
    while not eof do  
      begin  
        readln(a, b);  
        writeln('The GCD of ', a : 1, ' and ', b : 1,  
              ' is ', GCD(a, b) : 1, '.')  
      end  
  end .
```

The representation of the above documents can be graphically depicted as follows:



Note that the design document is a Z specification which is composed of a *collection* of Z paragraphs, including an axiomatic function description and a schema. The collective nature of this document segment is depicted by a box which has a label

`DesignDoc` at its top-left corner and encloses all its component segments. All other segments are sequential and are depicted in the usual tree form. Also note that some lower-level document segments are omitted from the representation, which is illustrated by dotted lines.

The syntactic structures of the above documents may be described using the augmented EBNF. The following are some of the EBNF productions, which we call *segment type rules*:

```

DesignDoc = Zspec ;
Zspec -C = { Zparagraph } ;
Zparagraph = ... | AxDesc | Schema | ... ;
AxDesc -S = [ Comment ] DecPart [ AxiomPart ] ;
Schema -S = [ Comment ] SchemaName DecPart [ AxiomPart ] ;
ImpDoc = Program ;
Program -S = [ Comment ] PrgHead Decs StPart "." ;
Decs -S = ... [ TypeDec ] [ VarDec ] { (ProcDec | FuncDec) } ;
FuncDec -S = [ Comment ] FuncHead Decs StPart ";" ;
StPart -S = "begin" Stmt { ";" Stmt } "end" ;
Stmt = ... | AssignSt | ... | WhileSt | ... ;

```

Note that the attributes `-C` and `-S` indicate that the segments being defined are collective and sequential segments respectively.

## 2.2 Relationships

As argued earlier, the relationships between software documents and those between their components should also be captured and represented. We do so by introducing structural relations between the relevant document segments. A *structural*

*relation* connects one document segment to another document segment. All the structural relations in a document space (i.e., all the documents concerned) are classified into different types, and each type of structural relations may connect certain types of document segments.

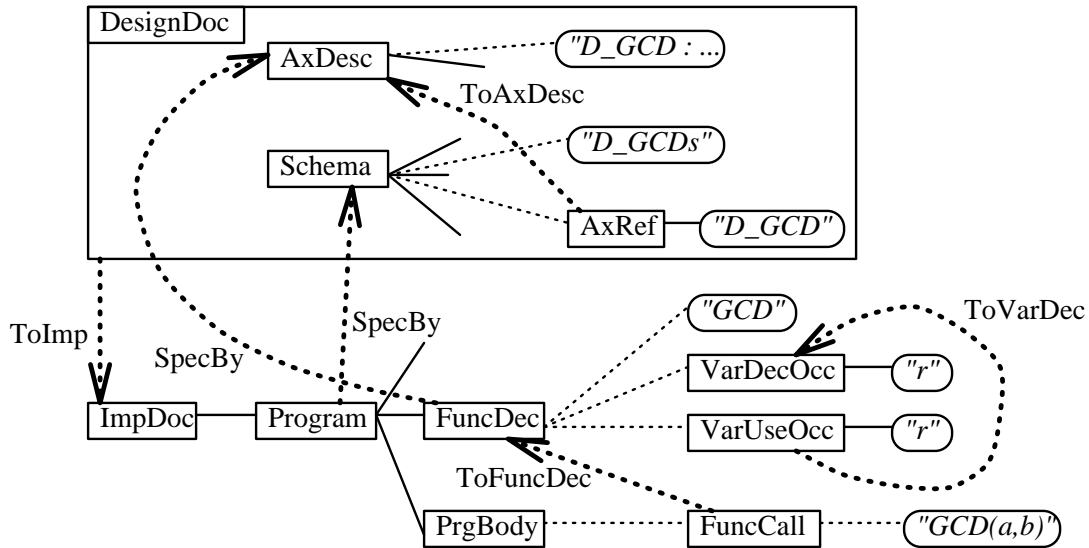
One class of relationships is the ones between entire documents, which we call *coarse-grained inter-document relationships*. In the GCD development, for example, the coarse-grained inter-document relationship between the design and implementation documents can be represented by a **ToImp** relation from the design document to the implementation document.

Another class of relationships is the ones between components of different documents, which we call *fine-grained inter-document relationships*. For example, a **SpecBy** inter-document relation can be used to represent the “specified-by” relationship from the Pascal function *GCD* in the implementation document to the Z function *D\_GCD* in the design document. Similarly, another **SpecBy** relation can be used to link the program *GCDs* in the implementation document to the schema *D\_GCDs* in the design document. In general, **SpecBy** relations can be used to represent the “specified-by” relationships from Pascal programs, procedures and functions to Z paragraphs.

The third class of relationships is the ones between components of same document, which we call *intra-document relationships*. In the Pascal function *GCD*, for example, a **ToVarDec** intra-document relation can be used to represent the relationship from the use occurrence of variable *r* in the assignment statement “*r* := *a mod b*” to its declaration in the variable declaration part. Other examples of intra-document structural relations include a **ToFuncDec** relation from the function call “*GCD(a, b)*” to the function *GCD*’s declaration in the implementation document, and a **ToAxDesc** relation from the reference of Z function *D\_GCD* in a predicate of the Z schema *D\_GCDs* to *D\_GCD*’s definition in the design document.

The example structural relations given above can be graphically depicted by the

arrows in the following representation:



The productions for describing structural relations are called *relation type rules*. They are composed of the type of the structural relation on the left hand side and the types of the connected segments on the right hand side. The type rules governing the above structural relations are:

```

ToImp = < DesignDoc , ImpDoc > ;
SpecBy = < Program , Zparagraph > | < ProcDec , Zparagraph > |
        < FuncDec , Zparagraph > ;
ToVarDec = < VarUseOcc , VarDecOcc > ;
ToFuncDec = < FuncCall , FuncDec > ;
ToAxDesc = < AxRef , AxDesc > ;

```

Inter- and intra-document relationships arise in different ways. Some, like the

traceability relationships between the different documents, are due to the software development methodology in use. Others, such as the relationship between an identifier use and its declaration, are more subtle and may be determined by analytic tools which understand the scope rules of the language concerned. In spite of such differences, they should all be captured in the documents' internal representation.

### 3 Structural properties of software documents

In the previous section, we have seen that software documents and their relationships are subject to certain representation constraints, which are expressed as the type rules for document segments and structural relations. In fact, the document segments and structural relations may have additional *structural properties*, which can be seen as additional *structural constraints* on the documents. To provide tool support for these properties, it is important that these properties can be specified precisely so that they can be used to define a structurally consistent document space. Before introducing our specification mechanisms for structural properties, we first have a look at the various inter-segment relations in the document space.

#### 3.1 Inter-segment relations

So far, we have seen two kinds of basic inter-segment relations: *parent-child syntactic relations* which are the relations from parent segments to child segments on the syntax trees of individual documents; *structural relations* which are as introduced in the previous section. With these relations and some relational operators, we can compose further relations (see below). Unlike the syntactic and structural relations, the *composed relations* are not introduced or deleted directly by the user or tools. But they can be used to state the structural properties of the documents.

To be able to formally state the composed relations, we introduce the following concepts and conventions:

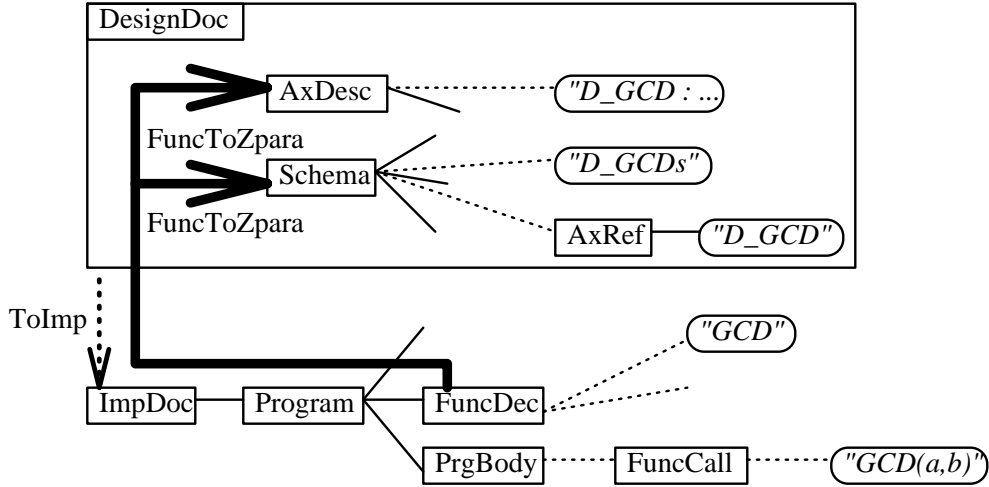
1. A *relation* is represented as an ordered pair of segments which are related by the relation concerned (from the first segment of the pair to the second segment of the pair).
2. A set of relations forms a *mathematical relation*.
3. We use **synRel** to denote all the parent-child syntactic relations.
4. We use **strucRel** to denote all the structural relations (irrelevant of their types).
5. We use *relType* to denote the set of relations which have the given type *relType*.
6. we use *segType* to denote the set of segments which have the given type *segType*.

Besides, we need a full range of relational operators. The following are some of these operators taken from the specification language Z [8]:

- Given a set *segst* of document segments,  $\text{id } segst$  evaluates to the mathematical *identity relation* with respect to *segst*.
- $R^\sim, R^+, R^*$ : inverse/transitive closure/reflective-transitive closure of  $R$ ;  
 $R_1 \circ R_2$ : relational composition of  $R_1$  and  $R_2$ ;  
 where  $R, R_1$  and  $R_2$  are relation expressions denoting sets of relations.
- $segst \triangleleft R, segst \triangleleft R$ : domain restriction/anti-restriction;  
 $R \triangleright segst, R \triangleright segst$ : range restriction/anti-restriction;  
 where *segst* is a set of document segments and  $R$  is a relation expression denoting a set of relations.

- Sets of relations are sets so that the following set operators apply:  
 $R_1 \setminus R_2$ ,  $R_1 \cup R_2$ ,  $R_1 \cap R_2$ : set difference/union/intersection;  
 $\cup\{g : gst \bullet R\}$ ,  $\cap\{g : gst \bullet R\}$ : generalised set union/intersection;  
 where  $gst$  is a set (of document segments, positive integers, and so on) and  $R_1, R_2$   
 and  $R$  are relation expressions denoting sets of relations.

With the conventions and relational operators introduced above, we can define composed relations. As an example, consider the composed relation (set) **FuncToZpara** from function declarations in the implementation document to Z paragraphs in the design document via the inverse of the **ToImp** relation:



where the firm arrows represent the composed relations of the type **FuncToZpara**. These composed relations can be defined as follows:

$$\text{FuncToZpara} = (\text{synRel}^+ \triangleright \text{FuncDec}) \sim \circ \text{ToImp} \sim \circ (\text{synRel}^+ \triangleright \text{Zparagraph})$$

where the part  $(\text{synRel}^+ \triangleright \text{FuncDec}) \sim$  describes the path(s) from **FuncDec** to **ImpDoc**,

the part  $\text{ToImp}^\sim$  leads to  $\text{DesignDoc}$ , and finally the part  $(\text{synRel}^+ \triangleright \text{Zparagraph})$  specifies the paths to  $Z$  paragraphs. In general, the composed relation is defined by specifying the required path(s) on the documents' representation graph (i.e., syntax trees plus structural relations). Note that a composed relation (set) can be used in defining other composed relations by referring its name.

### 3.2 Structural properties

The structural properties of software documents concern the document segments and structural relations in the document space. In general, these properties can be specified as Boolean expressions using some primitive predicates. In the rest of this section, we present a number of examples to demonstrate how various structural properties can be specified.

**Example 1.** In the context of the GCD development, consider the following property about the completeness of the document space:

*A complete development must have an implementation document.*

This property can be specified as follows:

$$\# \text{ImpDoc} > 0$$

where  $\# \text{ImpDoc}$  is the cardinality of the set  $\text{ImpDoc}$ .

**Example 2.** Consider the following property about the relationship between the implementation document and the design document:

*In a development, an implementation document must have a corresponding design document related by a  $\text{ToImp}$  relation.*

This property can be re-stated as that the set of **ToImp** relations is surjective with respect to all the **ImpDoc** segments:

$$\mathbf{surjective}(\mathbf{ToImp}, \mathbf{ImpDoc})$$

where **surjective** is a primitive predicate.

**Example 3.** Consider the following property about a Pascal program:

*Every function call in a Pascal program must have one and only one corresponding function declaration related by a **ToFuncDec** relation.*

This property can be re-stated as that the set of **ToFuncDec** relations is a total mathematical function relative to all the **FuncCall** segments:

$$\mathbf{total}(\mathbf{ToFuncDec}, \mathbf{FuncCall}) \wedge \mathbf{functional}(\mathbf{ToFuncDec})$$

where **total** and **functional** are primitive predicates.

**Example 4.** Consider the following property about the relationship between **SpecBy** relations and **ToImp** relations:

*For a **SpecBy** relation from segment  $A$  to segment  $B$ , there must be a **ToImp** relation from  $B$ 's root document (**DesignDoc**) to  $A$ 's root document (**ImpDoc**).*

To specify this structural property, we first define a composed relation which connects a program, a procedure declaration or a function declaration (in the implementation document) to a  $Z$  paragraph (in the design document) via the inverse of the **ToImp** relation:

$$\mathbf{PunitToZpara} = (\mathbf{synRel}^+ \triangleright (\mathbf{Program} \cup \mathbf{ProcDec} \cup \mathbf{FuncDec})) \sim \text{; ToImp} \sim \text{; } (\mathbf{synRel}^+ \triangleright \mathbf{Zparagraph})$$

(Note the similarity between `PunitToZpara` and `FuncToZpara`. In fact, `FuncToZpara` is a subset of `PunitToZpara`.) Then the above property can be re-stated as that the set of `SpecBy` relations is a subset of the set of `PunitToZpara` relations:

$$\text{SpecBy} \subseteq \text{PunitToZpara}$$

In general, a range of structural properties can be introduced for a software document space, and they can be quite complex. To a large extent, these properties capture the static semantics of the software documents and their relationships.

#### 4 Related work and discussion

In language-based editors, such as the Synthesizer Generator [6, 7] and the UQ editors [11, 1, 9], the representation of the documents concerned is usually based on ordered syntax trees. The inter- and intra-document relationships and the structural constraints on documents and relationships are not explicitly captured or stated. Rather, the relationships and consistency of documents are enforced by attribute calculations or checked by specific tools. In general, the structural properties of software documents in most language-based editors are stated or programmed in an operational style.

The PROGRESS approach to document representation and description is based on abstract syntax graphs [3]. It supports the capture and description of the syntactic structures of documents and the intra- and inter-document relationships. However, the structural properties are programmed in the graph-rewriting rules for document manipulation, and they are dynamically enforced in a consistency-preserving manner.

In the document representation and description approach investigated in the GOODSTEP project [2], the syntactic structures of documents are represented as abstract syntax trees. The support for the relationships and structural properties is

realised in the rules for accessing and modifying documents. Similar to PROGRESS, the document consistency is enforced and checked dynamically.

In our approach, the syntactic structures are represented by the traditional syntax trees, but with collective segments to reflect the un-ordered nature of certain segments. As in PROGRESS, we treat the inter- and intra-document relationships as first-class objects to facilitate traceability of the software development process. We have deployed a declarative style in stating structural properties so that a complete, static definition of a structurally consistent document space can be obtained without resorting to operation definitions.

The statically stated structural constraints are not enforced in a consistency-preserving manner, i.e., they can be violated in the document manipulation process. In practice, the user may choose to apply some or all of them to check the consistency of the documents at any desirable time. Our approach also allows dynamic consistency checking associated with operations for document manipulation, which will be reported in detail elsewhere. In general, we regard the separation between static description and dynamic programming for the structural properties as necessary because on the one hand we achieve a complete, static definition of the structurally consistent document space, and on the other hand we are able to implement flexible consistency strategies in operation definitions [5].

## 5 Conclusions and further work

We have argued that to facilitate the much needed traceability support for the software development process, the internal representation of software documents should capture the individual documents and the inter- and intra-document relationships, and support their structural properties. To provide such support, we have introduced an approach to specifying the structural properties of the documents and relationships in a declarative style. We have shown that the proposed approach is well

sued to the specification of a variety of structural properties. In particular, we have demonstrated that composed relationships can be used in stating these properties. The declarative style of these specifications makes it possible and easier to develop a complete structural definition for a document space without resorting to operations.

The representation and description issues reported in this paper are only part of a document model for software development environments that we have been developing. Other aspects of the model include document presentation [10] and document manipulation (to be reported elsewhere). Based on this model and a corresponding document description language, a generic environment kernel supporting software development is being developed.

**Acknowledgements.** Much of the work reported here was carried out when the author was with the University of Queensland. The author would like to thank David Carrington, Ian Hayes and Jim Welsh for their comments and discussions.

## References

- [1] B. Broom, J. Welsh, and L. Wildman. UQ2: A multilingual document editor. In *Proc. of 5th Australian Software Engineering Conf.*, pages 289–294, Sydney, Australia, May 1990.
- [2] W. Emmerich and P. Kroha. An object-oriented language for specification of integrated syntax-directed tools. Research paper, Department of Computer Science, University of Dortmund, Dortmund, Germany, 1993.
- [3] G. Engels, C. Lewerentz, M. Nagl, W. Schäfer, and A. Schürr. Building integrated software development environments Part I: Tool specification. *ACM Trans. on Software Engineering and Methodology*, 1(2):135–167, April 1992.

- [4] J. Han. Representation and description of software documents. To appear in *Proc. of IEEE Region 10's Ninth Annual Int. Conf. on Frontiers of Computer Technology*, Singapore, August 1994. IEEE Computer Society Press.
- [5] J. Han. Environment support for software consistency. In *Proc. of 1993 IEEE Region 10 Int. Conf. on Computer, Communication, Control and Power Engineering*, volume I, pages 399–402, Beijing, China, October 1993. International Academic Publishers.
- [6] T.W. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-based Editors*. Springer-Verlag, New York, 1989.
- [7] T.W. Reps and T. Teitelbaum. *The Synthesizer Generator Reference Manual*. Springer-Verlag, New York, third edition, 1989.
- [8] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall International, London, 2nd edition, 1992.
- [9] J. Welsh, B. Broom, and D. Kiong. A design rationale for a language-based editor. *Software – Practice and Experience*, 21(9):923–947, September 1991.
- [10] J. Welsh and J. Han. Software documents: Concepts and tools. To appear in *Software — Concepts and Tools*, 15(1), 1994.
- [11] J. Welsh, G.A. Rose, and M. Lloyd. An adaptive program editor. *Australian Computer J.*, 18(2):67–74, May 1986.