Type Modelling for Document Transformation in Structured Editing Systems

E. Akpotsui, V. Quint, C. Roisin

INRIA–IMAG
2, rue de Vignate – 38610 Gières – France
e–mail: extase@mururoa.imag.fr

Abstract:
This paper addresses the problem of type transformations in structured editing systems and proposes a type description model convenient for type comparison and document conversion. Two kinds of transformations are considered: dynamic transformations allow a structured editor to change the structure of a part of a document when this part is copied or moved, and static transformations allow specific tools to restructure documents when their generic structure is modified. We present in this paper the current state of our research on formal analysis for these transformations.

Keywords: structured document, type transformation, generic structure, document model.

1 Document transformation

1.1 Structured documents

During the last ten years, a large effort has been spent in developing systems for structured documents. As an example, document structure for interactive editing was one of the dominant themes of the EP’86 conference [26]. Seven years later, the promising technique of structured editing is not yet in common use. One of the reasons that can explain this delay is the constraints that a structured approach imposes to users.

In a structured editing system, a document is considered as a logical structure. It is made up of typed components such as title, abstract, section, paragraph, note, etc., which are assembled into a structure representing the document organization. The types of components and their relationships in the structure are defined by a generic structure and each document has a specific structure which is an instance of the generic structure. Several generic structures may be defined, for representing different types of documents: a letter
has not the same logical structure (types of components and relationships between components) as a report, which is different from a novel.

This approach has a number of advantages that are not discussed here (see [2] for more details), but it also has some drawbacks. One of them comes from the basic principle: each document must have a specific logical structure which is consistent with the corresponding generic structure. This implies that any change in a document can be done only if the generic structure allows it, which is often considered by users as a very strong constraint.

This paper considers the most frequent problems to which a user is faced when he wants to modify documents in a structured editing system. Most of these problems have been identified by users of the Grif system [21] [11] and the proposed solutions are implemented in Grif, but it is worth noting that the same problems arise with standards such as ODA [17] or SGML [14], [16], as these standards are based on the same notion of a generic structure (called a Document Type Definition in SGML). Throughout this paper, the terms generic structure and type are interchangeably used.

Grif is an interactive system for editing and formatting complex structured documents and it follows the principles stated above. The specific logical structure of a document is described by a generic logical structure, corresponding to a SGML DTD with minor differences such as the set of available basic types; in Grif, basic elements can be not only of type text but also of type, graphic or picture. The consistency of the specific structure is guaranteed by the editor, which guides and controls the user according to the generic structure of the document being edited. In particular, the editor prevents the user from producing a document whose specific structure would not be consistent with the generic structure.

1.2 Needs for document transformation

In most interactive document preparation systems, a command allows the user to copy or cut a part of a document (the source) and to insert (or paste) it in some other part (the target) of a document. A system based on a structured representation of documents must take into account the type of elements constituting the source part and the type of the target. These parts may have the same type or different types. The strong typing of document components often leads the system to reject the command, when types are different; this is because the editor strictly checks the pasted structure and refuses it if it is not compatible with the target.

This problem can be considered either as integrating the source structure in a given element (the target), or as creating a new element from the source. In both cases the user would like the operation be possible, whatever the type of the involved elements. The problem becomes more complex when source elements come from a document and the target is in another document. In that case, the two documents can be constructed according to different generic structures.
For solving this problem, the structure of source elements must be transformed to become consistent with the generic structure of the target. Most users want this transformation to be automatically performed while they are editing a document, but they accept to indicate their preferences when several transformations are possible. This type of transformation performed by an interactive editor is called a *dynamic* transformation.

Dynamic transformations are also needed when a user wants to restructure an existing, structured (or even unstructured) document for organizing it in a different structure. With currently available systems, the most frequent solution consists in creating a new, empty structure and to fill it manually with the contents of the old structure, what is a very tedious task.

Editing commands are not the only cases of structure transformation. Transformations are also required by changes in the generic logical structures. The logical structure of a document type can evolve. For various reasons it may be necessary to declare new elements, to remove elements that have become useless in some type of document, or to arrange existing elements in a different order. These changes lead to new versions of generic structures and the user has to specify into which new type each old type has to be transformed.

The problem is then to recover documents built with old versions of a generic structure that has evolved. As a number of such documents may exist, it is necessary to transform them automatically, for making them consistent with the new generic structure. This kind of operation is called a *static* transformation because it is usually performed outside an editing session.

### 1.3 Identifying kinds of structure transformations

The general problem addressed by both static and dynamic transformations is to restructure a document instance (or a part of a document instance) conforming to a given generic structure (a given type) into an instance conforming to another generic structure (another type). Two types are involved: the source type and the target type.

In dynamic transformations (for example, in a paste operation), the source type is the one of the element that has been cut or copied and that is intended to be pasted somewhere else. The type of the element into which the source element is to be pasted is the target type. The integration of the content of the source element’s content in the target is called *adoption* and the target type is said to *adopt* the source type.

In a static transformation, the source type is the type of the document to be transformed, and the target type is the new type to which the document must conform. In Grif, we have developed a tool performing static transformations. This tool, described in [1], works in two steps (see Figure 1):

- A first program, called the Comparator, produces transformation rules by comparing the two types involved.
- A second program, called the Converter applies these rules to document instances, in order to transform them.

The comparison is based on the detection of all changes in the definition of each type of the generic structure. Comparison is performed using type homonymy (a type of the source generic structure is transformed into the type of the same name in the target generic structure), extended with explicit type correspondences expressed by renaming rules. For each change detected, transformation rules are produced and are attached to the source type involved.

![Diagram](image)

*Figure 1: Static transformation in Grif*

An automatic transformation process can lead to an undeterminate choice of the target types because the user may not give a complete set of renaming rules. It is necessary to understand exactly what the user’s intention is when a transformation of a type $t$ into a type $t'$ is requested:

1. Is the respect of the target structure of higher priority than the recovery of the source element content (i.e. the information contained in the basic elements)?
2. Has the component order in the source to be preserved in the target?
3. If the type of a source element has no corresponding type in the target type, has this element to be suppressed? Could it be possible to find a *compatible type* in order to perform the transformation without losing information?
4. What properties of types are involved in the compatibility relations for type transformations?
The answers to these questions are not the same for static and dynamic transformations. Therefore, the methods that have been implemented in Grif for static transformations have not been considered suitable for dynamic transformations. The problem is that, in dynamic transformations, the focus of interest is not the structure but the content of the source element, which should be preserved even if the structures of the source and target elements are not identical. Since static transformations are mainly based on structure comparison, this could lead to the rejection of adoption, thus losing the content of the source. Moreover, even during a static transformation, the adoption process may be necessary when transformation rules cannot be directly deduced from renaming rules, homonymy of types names and change analysis.

The purpose of our work is to define relations between types that could help implementing transformation operations. This paper presents a formalism for type comparison and its applications to structured document transformations.

The next section states the notion of generic structure through an example and defines the mapping between document types and a tree model. In section 3, we associate functions to types in order to introduce the properties of types that are necessary to express strong equivalence for structural transformations. We show then that these functions allow to express all elementary changes that can appear during transformations. In section 4, some relations between types are proposed; these relations will allow different degrees of flexibility in type transformations, corresponding to different adoption strategies. Section 5 is devoted to a specific transformation method that is better adapted to dynamic transformations: the key idea is to consider the leaves of a type tree as part of an alphabet and to use the language built on this alphabet to provide transformations solutions that preserve the whole original information. The last sections present related work and give some conclusions.

2 A tree model for document types

In structured editing systems, the logical structure of documents can mainly be considered as a tree structure in which each node has a type. Figure 2 shows the tree structure of a document of type Article, which is defined in section 2.1. Our aim here is to show that not only document instances, but also generic structures, as defined by DTDs, can be considered as trees.

First we present an example of DTD and its corresponding tree representation: it will define the vocabulary used and the limitation of our study. For the sake of simplicity, the generic structures presented in the following are written using the SGML syntax augmented with the Grif basic types, which are not available in SGML. We then define the tree model more formally.
2.1 Document type definition

A generic structure is a set of types constructed with a set of basic types called $\beta$ and a set of constructors called $C$:

- $\beta = \{\#PCDATA, PICTURE, GRAPHIC\}$, where $\#PCDATA$ represents a character string, $PICTURE$ a bit-map or a photograph and $GRAPHIC$ a single geometric shape such as a rectangle or a circle,
- $C = \{\text{Ordered Group, Unordered Group, List, Choice, Identity}\}$.

The following example is the generic structure (or DTD) of an Article. It gives the correspondence between these constructors and the SGML notation.

```xml
<!ELEMENT Article (Front?, Title, Author, Body, Appendix) >
<!ELEMENT Front (Date & Number & Version?) >
<!ELEMENT Title | Author | Date | Number | Version (#PCDATA) >
<!ELEMENT Body (Chapter)+ >
<!ELEMENT Chapter (Title, Section+) >
<!ELEMENT Section (Title, ParaList, SubSectList?) >
<!ELEMENT ParaList (Para)+ >
<!ELEMENT Para (SimplePara | ElList) >
<!ELEMENT SimplePara (#PCDATA) >
<!ELEMENT ElList (ElChoice)+ >
<!ELEMENT ElChoice (PICTURE | GRAPHIC | Para) >
```

Figure 2: Partial tree representation of a document instance of type Article
In this example, Article is an Ordered Group (connector ','), Front is an Unordered Group (connector '&'), Body is a List (occurrence indicator '+'), and Para is a Choice (connector '|'). Version is an optional component of Front (optional indicator '?') and indicator '-' expresses excluded types (PICTURE cannot appear in Appendix).

Let $T_t$ be the set of the types defined in the generic structure $t$, $T_t = \{ u: u \in \beta$ or $u$ is a constructed type $\}$. In order to have a bijection between the elements of set $T_t$ and the set of the nodes of the corresponding tree representation introduced in section 2.2, the generic structure is transformed into a canonical form by the following operations:

- Parameter entities are replaced by their definitions (SGML parameter entities are similar to macros).
- A type name is given to intermediate types. For example, the following definition of type Chapter:

  ```xml
  <!ELEMENT Chapter            − −     (Title, Section+)                                         >
  ```

  is transformed into:

  ```xml
  <!ELEMENT Chapter            − −      (Title, SectionList)                                    >
  <!ELEMENT SectionList       − −     (Section+)                                                  >
  ```

- Constructed types used more than once the DTD are renamed. For example, if $x$ is defined by the Ordered Group $(y, y, z)$, $y$ must be renamed in order to have $(y_1, y_2, z)$. Another example is given in Figure 3 with the recursive type Para which is renamed when it is used in ElChoice:

  ```xml
  <!ELEMENT ElChoice          − −         (PICTURE | GRAPHIC | Para)           >
  ```

  becomes:

  ```xml
  <!ELEMENT ElChoice          − −         (PICTURE | GRAPHIC | Para1)         >
  ```

  In a similar way, each use of the type Title is renamed.

### 2.2 Tree definition of document types

In this section, we first introduce the notation for representing trees and then we define a tree representation for document types.

#### Tree definition

A tree is a couple $(S, p)$ where $S$ is a set and $p$ an application of $S$ into $S$ such that there exists an element $r$ of $S$ respecting the two following properties:

1. $p(r) = r$,
   - $r$ is a distinguished element of $S$, known as the root of the corresponding tree, hence the fixed point of the function $p$.
2. $\forall x \in S, \exists k \in \mathbb{N} : p^k(x) = r$
A tree is a connected graph with no cycle, and ordered by $p$. The following vocabulary is used in the rest of this article:

- **y is the parent of x** $y = p(x)$ and $y \neq x$
- **x is a child of y** $y = p(x)$ and $x \neq y$
- **children of y** \( c(y) = \{ x \in S : y = p(x) \} \)
- **ascendants of x** \( A(y) = \{ y \in S : \exists k \in \mathbb{N}, y = p^k(x) \} \)
- **descendants of y** \( C(y) = \{ x \in S : \exists k \in \mathbb{N}, y = p^k(x) \} \)

**Semantics of the tree definition for document types**

Let $t$ be the document type (in a canonical form) to be represented as a tree. Consider for example that $t$ is Article as in the example of Figure 3. The set of tree nodes is $T_t$, which is the set of type names used in the definition of $t$.

The function $p : T_t \rightarrow T_t$ expresses that $\forall x \in T_t$, $p(x)$ is the type in the definition of which $x$ appears in the canonical form of the DTD.

The leaves of the tree ($T_t$, $p$) are basic types or recursive types. In a DTD that is not in a canonical form, a recursive type is a type that appears at least twice when following a sequence of definitions, such as Para, ParaList, ElChoice, Para; in that example, Para is a recursive type. In a canonical DTD, renaming avoids recursion and the renamed recursive types become leaves. Let $R_t$ be the set of recursive types that appear in the document type $t$.

In the next section, other applications on nodes are defined in order to express other characteristics of DTD such as the name of the constructor, optional or mandatory types, exclusion, cardinality of a List, etc.

**Example**

Figure 3 shows a part of the tree corresponding to the DTD Article given above. For the sake of simplicity, all nodes are not represented, neither other information on types like constructors, optional elements, etc.
Type tree and instance tree

The main differences between the tree of a document instance (Figure 2) and the tree of a document type (Figure 3) are the following:

1. A type being a generic structure, it represents the family of the possible instances that can be built from it. Hence, the tree representing a document type comprises all its possibilities. As an example, a type whose constructor is a Choice is made up of options. Its corresponding tree shows all options as children, as shown for the type ElChoice in Figure 3. The semantics related to a Choice state that an element of such a type must have only one child whose type is one of the options of the Choice. Therefore, an instance of type Choice has only one child, as shown for the element of type ElChoice in Figure 2.

2. The type tree corresponding to a Group with optional types includes all these types as children while a corresponding instance tree may not comprise optional elements. In this case, the instance tree doesn’t map exactly its type tree. For example the document instance on Figure 2 has no element of optional type Front.

3. In a recursive type tree, any occurrence of the recursive type as a descendant is a leaf (for example Para1 in Figure 3), restricting the depth of branches that contain

Figure 3: Partial tree representation of the DTD Article
such occurrences. On the contrary, there is no depth limitation to the recursive branches in the instance trees.

3 Type definition for static analysis with strong structural constraints

We first define the functions that are necessary for comparing types, and then we indicate how type changes for static transformations are expressed in this formalism. The goal is to obtain comparison criteria that are richer than the simple equivalence of type names.

A document type is not only characterized by the structural position of each of its components (which is given by the tree representation), but also by its constructor. Additional information is needed for a complete type analysis: rank of types in an Ordered Group, optional or mandatory indicators and exclusion. These attributes are given by functions whose domain is the set $T_t$. So on each node of a tree representing a type, the following functions are needed (they are defined in the following):

1. The types used to define another type (children of $t$): $c(t)$
2. The constructor of a type (&, | in SGML): $\text{cons}(t)$
3. The cardinal of a type: $n(t)$
4. The rank of types in type definition: $r(t)$
5. The excluded elements (− in SGML): $x(t)$
6. The optional elements (?) in SGML): $o(t)$

3.1 Functions on types

Types that define $u$: $c(u)$ and $\text{recurs}(u)$

By construction of the tree representing a type $u$, the set of types that define $u$ is given by the function $c(u)$ defined in section 2.2; more precisely:

1. $c(u)$ is the set of types which appear syntactically in the definition of $u$. Example: $<!ELEMENT Chapter \* (Title, SectionList) >$
   $$c(\text{Chapter}) = \{\text{Title, SectionList}\}$$

2. Basic types have no children:
   $$u \in \beta \iff c(u) = \emptyset$$
   Example: $c(#\text{PCDATA}) = \emptyset$

3. A recursive type appears as a leaf:
   $$u \in R_t \iff c(u) = \emptyset$$
   For these types, the function $\text{recurs}$ is defined: $T_t \rightarrow T_t$:
recurs(u) = v, where v is the type defining this recursive type. In other words, recurs(u) returns the original name for a recursive type that has been renamed.

This function is necessary to allow the comparison of recursive types.

In the example of Figure 3, para1 ∈ Rₜ, and recurs(Para1) = Para.

The function c defines an equivalence relation $R_c$ between types:

$$t_1R_c t_2 ⇔ c(t_1) = c(t_2)$$

This relation will be used to allow the adoption operation during transformation of documents: the insertion of an element (part of document) of type $t_1$ into a document accepting type $t_2$ is possible when types $t_1$ and $t_2$ are equivalent according to that relation.

**Constructor of a type: cons(u)**

The constructor of a type is given by the function cons:

$$T_t → C : \forall u ∈ T_t, cons(u) is the constructor used in the definition of u.$$  

Example: `<!ELEMENT Chapter − − (Title, SectionList)>`  
cons(Chapter) = Ordered Group

**Cardinal of a type: n(u)**

$n(u)$ is the maximum number of children of an instance of type $u$.

1. $cons(u) ∈ \{Ordered Goup, Unordered Group\} ⇒ n(u) = #c(u)$
2. $cons(u) = List ⇒ n(u) = ∞$
3. $cons(u) = Choice ⇒ n(u) = 1$
4. $cons(u) = Identity ⇒ n(u) = 1$

Examples:
- $n(SectionList) = ∞$
- $n(Para) = 1$
- $n(Chapter) = 2$

**Rank of the types of c(u): r(u)**

If $cons(u) = Ordered Group$, $r(u)$ is the bijection:

$c(u) → [1, #c(u)] : \forall u ‘ ∈ c(u), r(u)(u’) = rank of u’ in the definition of u,$  
where in the case of an Ordered Group, $#c(u) = n(u)$.

Example: `<!ELEMENT Author − − (Name, Affiliation)>`  
$r(Author) = \begin{cases}  
Name & → 1 
\text{Affiliation} & → 2 
\end{cases}$
Optional types: $o(u)$

The optional types of a definition of a type $u$ is a subset of $c(u)$ given by a function called $o : T_t \rightarrow P(T_t)$.

Example: `<!ELEMENT Front − −  (Date & Number & Version?)  >`  
\[ o(\text{Front}) = \{\text{Version}\} . \]

Exclusion of a type: $x(u)$

The exclusion of a type $u'$ from a type $u$ means that type $u'$ cannot be used at any level in the instances of $u$. As the function $c$ corresponds only to the first level definition of a type, another function is needed, $x : T_t \rightarrow P(T_t)$ that defines by $x(u)$ the types excluded from a type $u$.

Example: `<!ELEMENT Appendix − −   (Title, ElList, SubSect?) −(PICTURE)    >`  
\[ x(\text{Appendix}) = \{\text{PICTURE}\} . \]

3.2 Comparing types

Let $T_t$ be the set of types defined in the generic structure $t$ of documents to be transformed (the source type) and $T_{t'}$ the set of types defined in the generic structure $t'$ to which documents transformed must conform (the target type). In static transformations, the Comparator has to know in which target type $u' \in T_{t'}$ each type $u \in T_t$ has to be transformed: this is given by an association function $a: T_t \rightarrow T_{t'}$.

Example:

```
<!ELEMENT Section          − −   (Title, ParaList, SubSectList?) >
<!ELEMENT NewSection   − −   (Title, ParaList)                         >
```
\[ a(\text{Section}) = \text{NewSection} \]

With the above definitions, a type is modelled by the $n$–uplet $(t, c(t), cons(t), r(t), o(t), x(t))$. This model is used in the Comparator for identifying basic transformations of types. The basic transformations considered are the following (they are defined below):

1. restriction (Group, Choice),
2. reordering (Group),
3. constructor change (Group, List),
4. exclusion and restoration of excluded elements,
5. extension (Group, Choice),
6. mandatory (Optional),
7. cardinal reduction (List).
Restriction
Type restriction between t and a(t) is identified by: \(\exists u \in c(t) : a(u) \notin c(a(t))\).
For example, the transformation of Section into NewSection implies a restriction:
\[c(\text{Section}) - c(\text{NewSection}) = \{\text{SubSect}\}\]
The operations to be performed by the Comparator when a restriction is identified are:
1. Generation of a restriction rule.
2. Generation of inferred adoption rules of instances of type
   \(u \in c(t) - c(a(t))\)

Reordering
Reordering is identified when:
\[
\text{if } \text{cons}(t) = \text{cons}(a(t)) = \text{Ordered Group} \text{ and } \forall u \in c(t), a(u) \in c(a(t)),
\exists u \in c(t) : r(t)(u) \neq r(a(t))(a(u))
\]
The new rank of u is \(r(a(t))(a(u))\).
The rule produced by the Comparator is a translation rule for elements of type u.
Figure 4 illustrates this kind of transformation:

![Figure 4: Reordering elements in an Ordered Group](image)

Constructor change
Constructor change is identified when: \(\text{cons}(t) \neq \text{cons}(a(t))\).
Such transformations rely on the semantics of the constructors involved. Usually,
no elementary transformation can be performed when constructors are different and
other solutions must be considered (cf. section 4).
However, some simple cases can be mentioned:
- \(\text{cons}(t) = \text{Ordered Group} \text{ and } \text{cons}(a(t)) = \text{Unordered Group}\).
- \(\text{cons}(t) = \text{Unordered Group} \text{ and } \text{cons}(a(t)) = \text{Ordered Group}\) if the order of
  the elements is not considered.
- \(\text{cons}(t) = \text{Group with } n(t) = 1 \text{ and } \text{cons}(a(t)) = \text{Identity or List}\).
Exclusion

Exclusion is identified when:

1. \( \forall u \in T_t : u \in x(t) \text{ and } a(u) \notin x(a(t)) \text{ and } a(u) \in T_{a(t)} \),

   \( a(u) \) must be restored (if not optional).

2. \( \forall u \in T_t : u \notin x(t) \text{ and } a(u) \in x(a(t)) \).

   Instances of type \( a(t) \) must not contain elements of type \( a(u) \), so they must be removed if they exist in the source instance.

In order to lose no information, an adoption rule may be generated. As elements of type \( a(u) \) must not be created — to respect the definition of \( a(t) \) — possible solutions are (cf. section 4):

- search for equivalent elements,
- search for a cluster,

Type extension and mandatory types

Type extension is identified when: \( \exists u' \in c(a(t)) : \forall u \in c(t), a(u) \neq u' \).

Mandatory types in \( a(t) \) occur when \( \exists u \in o(t) : a(u) \in c(a(t)) \text{ and } a(u) \notin o(a(t)) \).

In both cases, the Comparator generates a creation rule (even if there is no content to put into).

4 Types equivalence for complex transformations

The application of elementary transformation rules is not satisfactory because it is based on type homonymy. Even in strongly constrained conversions, type homonymy can discard valid type transformations when the two types involved (or their descendants) have the same definition but not the same name.

Transformations that are not covered by elementary transformations are:

- static adoption (inferred adoption rule),
- dynamic adoption (cut and paste),
- change of type requested by the user during an editing session.

Relations between types are defined to enable such transformations. These relations identify many levels of structural differences between types, which allow less and less constrained transformations or correspond to different semantics required by the user:

- The target type and the source type must have the same name (type homonymy).
- The target type must have the same structure as the source type (type isomorphism, defined in section 4.1) or a structure close to it (constructor compatibility).
• The target type must contain the source type, i.e. the source type tree is a subtree of the target type tree; a subtype relation is defined in section 4.2 for that purpose.
• The target type must contain the source type, but in a broader sense; a cluster relation is defined for that in section 4.3.
• There is no structural constraint but the whole information of the instances of the source elements must be preserved in the target instances (in the same order or even in a different order); this case is presented in section 5.

4.1 Type isomorphism

Two types $t$ and $t'$ are isomorphic or structurally equivalent, which is noted $t \equiv t'$, if and only if all the following properties are verified:

1. The tree structures are identical: there is a bijection $\varphi$ between the set of nodes $T_t$ and $T_{t'}$ of the two trees representing respectively type $t$ and type $t'$:
   
   \[ \varphi : T_t \rightarrow T_{t'} : \varphi \text{ is bijective, and } \forall x \in T_t, \varphi(p(x)) = p(\varphi(x)). \]

2. $x$ and $\varphi(x)$ have the same constructor: $\text{cons}(\varphi(x)) = \text{cons}(x)$.

3. $x$ and $\varphi(x)$ have the same maximum number of children: $\forall x \in T_t, n(\varphi(x)) = n(x)$.

4. The types that are children of $x$ and of $\varphi(x)$ are equivalent:
   
   $\forall x \in T_t, \forall y \in c(x), y \equiv \varphi(y)$.

5. The optional types in $x$ and $\varphi(x)$ are equivalent:
   
   $\forall u \in o(x), \exists v \in o(\varphi(x)) : u \equiv v$.

6. The types excluded from $x$ and $\varphi(x)$ are equivalent:
   
   $\forall u \in x(x), \exists v \in x(\varphi(x)) : u \equiv v$.

In conclusion, all functions are equal or equivalent, but the names of the involved types are different.

This equivalence relation, $t \equiv t'$, expresses the identity of structure between the types $t$ and $t'$ and allows the transformation of any instance of type $t$ into type $t'$ (and vice versa). It defines equivalence classes on the set of all types managed by the system.

Example

In the following type definition:

```xml
<!ELEMENT ParaList − − (Para)+ >
<!ELEMENT Para − − (SimplePara | ElList) >
<!ELEMENT PLList − − (P)+ >
<!ELEMENT P − − (SimplePara | ElList) >
```

types ParaList and PLList are isomorph. Figure 5 shows their corresponding trees.
The algorithm that checks the equivalence between ParaList and PList verifies that Para is equivalent to P (since they have the same tree structure and their children have the same types) and then that ParaList is equivalent to PList (since they have the same tree structure and their children are equivalent).

**Type compatibility**

The equivalence classes are based on structural identity. These classes can be extended if a relation of compatibility, depending on the constructors defining the types, is used instead of the structural identity.

The advantage of this extension is to increase the number of allowed type conversions. The notion of compatibility is presented through two examples:

**Example 1:**

\( t \) is compatible with \( t' \) (Figure 6), if the following properties are verified:

1. \( \text{cons}(t) = \text{Group} \) and \( \text{cons}(t') = \text{List} \) and \( \text{cons}(c(t')) = \text{Choice} \)
   (Group may be an Ordered Group or an Unordered Group),
2. \( n(t) = n(t') \),
3. \( c(t) = c(c(t')) \), which means that the items of the Group are the options of the Choice.

The consequence is that an instance of type \( t \) can be converted into an instance of type \( t' \), but the reverse depends on the component types of the List instance.

**Example 2:**
Similarly, if $t$ and $t'$ are defined as a hierarchy of Group constructors, we can transform $t$ into $t'$ (and vice versa) if the leaves of the two trees have the same type (or are isomorph). Figure 7 illustrates this kind of compatibility.

![Figure 7: Type compatibility between Group types](image)

### 4.2 Subtypes

The notion of subtype defined here is based on the notion of subtree. So it is different from subtyping used in programming languages.

Type $u$ is a subtype of type $t$, denoted by $u \ll t$, if and only if:

$$\exists v \in T_t, u \equiv v.$$  

A consequence of the property that $u \ll t$ is that:

1. $u$ can be adopted by $t$,
2. $u$ can be transformed into $t$,
   but other parts of the tree $t$ besides $v$ must be created, even empty.

**Example**

In the example of Figure 5, we can see that:

Para $\ll$ PList, because Para $\equiv$ P, with P $\in$ T_PList.

### 4.3 Type clusters

The concept of type cluster is a generalization of the the notion of subtype, since it can be considered as a set of subtypes [10].

Let $t$ and $u$ be two types; $u$ is a cluster of $t$ if and only if the following conditions are satisfied:

1. $T_u \subseteq T_t$, every node of the cluster tree $u$ belongs to tree $t$.
2. $\forall x \in T_u$:
   - $p_u(x) \in A_t(x)$,
     the parent of $x$ in the cluster $u$ is an ascendant of $x$ in the tree $t$.  

• $\forall y \in A_t(x) : y \in A_u(x)$, $\forall z \in C_u(x), y \in A_u(z)$

every ascendant of node x (in tree t) which belongs to the cluster is also an ascendant of its descendents in the cluster u.

**Example:**
Suppose we have the following type definition:

```xml
<!ELEMENT OldChapter       − −       (Title, ParaList)       >
```

If a(OldChapter) = Chapter, OldChapter is a cluster of Chapter because every node of OldChapter belongs to Chapter and the parenthood order is respected (see Figure 8). So it is possible to transform an instance of OldChapter into an instance of Chapter (the complement subtrees must eventually be created). Several transformations may be possible; this problem is addressed in section 5.4.

![Figure 8: Example of a cluster](image)

Figure 8: Example of a cluster

## 5 Tree model for dynamic transformation

During cut and paste operations, the user’s concern is to keep the content of the source element. This must be seen as a requirement of higher priority than preserving the structure, and it highlights the difference between static and dynamic transformations. The semantics we use for the cut and paste operations are the following:

- Preserving the content of the source element is the main goal of the operation.
- The resulting structure must conform to the type definition of the document.
- Preserving the source structure is not required.
- The existence of a target structure identical to the source structure represents the best and simplest condition for a paste operation.
Two complementary approaches are investigated: the top-down approach and the bottom-up approach.

- The first approach is based on tree analysis and on the type model defined in the previous sections. The difficulty is to identify the best structural position for inserting the nodes of the source structure into the target tree: either the target node (or a descendant) admits the source tree as a child, or the insertion can only be done in breaking down the source tree (the source tree is then considered as a potential tree cluster of the target). In this latter case, the algorithm of tree comparison is based on the recognition of tree clusters (section 4.3).

- The second approach is based on the analysis of the leaves of the instance tree to be pasted.

As the second approach appears to be easier to implement, it has been studied first, and we develop only that approach in the following.

5.1 The language of a type

The approach is illustrated by examples based on the document structure presented in Figure 9. To each type we associate a language whose alphabet is the set of the basic types used in the definition of this type. Each element (word) of this language is called a base of the corresponding type, since the whole structure of the type can be generated from this base. The computation of this language is based on the concatenation operation (represented by a dot) and on the order of the components of a type. So, the elements of the alphabet can be set at their right position in a word.

Let the definition of type Section be the following:

```xml
<!ELEMENT Section                    − −             (SectTitle, SectBody, SubSection?)    >
<!ELEMENT SectTitle                  − −             (#PCDATA)                                          >
<!ELEMENT SectBody                 − −             (TextElem | Figure)                              >
<!ELEMENT SubSection               − −            (SubSectTitle, SubSectBody)              >
<!ELEMENT SubSectTitle            − −            (#PCDATA)                                          >
<!ELEMENT SubSectBody           − −            (TextElem | Figure)                               >
<!ELEMENT TextElem                  − −            (#PCDATA)                                          >
<!ELEMENT Figure                       − −             (FigTitle, Illustration)                           >
<!ELEMENT FigTitle                     − −            (#PCDATA)                                          >
<!ELEMENT Illustration                − −            (PICTURE | GRAPHIC)                       >
```

Figure 9 shows the tree representation of an instance of that type. According to the definition of the language of type Section, the word corresponding to this instance is #PCDATA.#PCDATA.#PCDATA.#PCDATA.PICTURE.
Here are some examples of languages associated to types corresponding to the example of type Section. Let \( L \) be a function that returns the language associated to a type:

\[
L(x) = \emptyset, \quad \forall \ x \in \beta.
\]

\( L(\text{SectTitle}) = \{\#\text{PCDATA}\} \).

\( L(\text{Illustration}) = \{\text{GRAPHIC}, \text{PICTURE}\} \).

\( L(\text{Figure}) = \{\#\text{PCDATA}.\text{GRAPHIC}, \#\text{PCDATA}.\text{PICTURE}\} \).

\( L(\text{SectBody}) = \{\#\text{PCDATA}, \#\text{PCDATA}.\text{PICTURE}, \#\text{PCDATA}.\text{GRAPHIC}\} \).

A type may have more than one word in its language. This is due to the Choice constructor and the optional operator, as illustrated by \( L(\text{Illustration}) \). Moreover, the language generated may be infinite, due to the List constructor and the recursive type definitions.

### 5.2 The paste operation

Since the source and target elements involved in the paste operation have their own type, each one has its own language. The steps of the execution of the paste operation are the following:

1. Locate the word in the source element type language corresponding to the source element.
2. Search for matched patterns in the target language.
3. Search in the matched patterns for the ones whose structure is identical or closer to the source element structure. The existence of a matched pattern does not mean that the structures involved are identical. The problem of choosing for the best structure to make the adoption is addressed in section 5.4.
4. Make the target element adopt the source element.
5.3 Application in the Grif editor

As well as most structured document editors, the first versions of the Grif editor were able to perform the paste operation only in the case where the types of the elements involved in the operation were identical. As an example, when cutting SubSection of Figure 9 and asking the editor to paste it as a new Section, it generated a new Section into which the source SubSection were pasted as a SubSection, as shown in Figure 10. But this is not the result expected by the user, who wants the children elements of SubSection, elements SubSect and SubSectBody, to be respectively transformed into the new elements SectTitle and SectBody.

![Diagram](image)

*Figure 10: Result of a dynamic transformation based on type identity*

The word in L(SubSection) related to the source instance SubSection to be pasted is #PCDATA.#PCDATA.PICTURE. This word belongs to L(SectTitle).L(SectBody). Hence, the paste operation is performed as follows:

- A new element Section is created as asked by the user.
- The target element of type SectTitle adopts the source element of type SubSectTitle. The target element is attached as a child of the new Section.
- A new target element of type FigTitle adopts the source element of type FigTitle.
- A new target element of type Illustration adopts the source element of type Illustration.
- Both FigTitle and Illustration are attached as children of a new element of type Figure that is hooked to a new element of type SectBody which at last is hooked to the new Section.

This results in the adoption of the source element of type SubSection by an element of type Section, as illustrated in Figure 11. In other words, the element of type SubSection has been transformed into an element of type Section, that has no element of type SubSection contrary to the result achieved in Figure 10.
5.4 Multiple solutions for adoption

Another aspect of the problem is the existence of more than one solution, what poses the problem of choosing the best solution to the transformation request. Below are two examples:

**Multiple matched words**

The word associated with the source element can match several substrings in the target language, making it necessary to search in the target tree for the subtree whose structure is closer to the source structure. The criterion of such a choice ranges from type equivalence to type compatibility.

**Multiple matched clusters**

The criterion that is under consideration for the choice of the best cluster is the dimension of the cluster, based on the linear order that characterizes a tree. This dimension is supposed to give the exact distance between two nodes and to express how the whole cluster is interspersed in the target structure. It seems clear that the most compact cluster, that is the cluster with the least dimension, will be the most suitable.

6 Related work

Structured editing systems are not the only systems faced with problems of type conversions; at least three other areas deal with these problems: programming languages, structure–oriented programming environments and object–oriented databases.
6.1 Structured editing systems

As shown in the above examples, type formalization is a requirement for structured document editors to provide users with convenient and flexible tools. However this area has not yet been deeply investigated and current work on document processing models is mainly oriented towards document formatting, like in [5] and [15]. Nevertheless, a few proposals have been made for solving the problem of type transformations in structured document editors.

The proposal by Furuta and Stotts [12] uses the theory of hierarchical graphs for relating two structures, the source and target types involved in the transformation. This study leads to two classes of changes called atomic transformations and composite transformations and both are what we call elementary transformations.

Another solution has been proposed for the Rita editor [9] to handle simple type conversions. In this system, a type t is converted into a type t’ only if the ordered set of the children of t is part of the ordered set of the children of t’. In the case of an unsuccessful automatic paste request, the user can make a manual transformation of the offending source element, by editing it in a patch area where type checking is relaxed. The new manually transformed element can be pasted in the document if it causes no unrecoverable type inconsistency in the target element.

An improvement of these limited type conversions is proposed in [8] to support the cut and paste operations. It is based on the augmentation of the document type definition by a list of couples of types, called fallback list. In a fallback statement, a type is related to the type into which it can be converted: \( \text{par} \rightarrow \text{spar} \) is a fallback statement meaning that an element of type \( \text{par} \) (paragraph) can be converted into an element of type \( \text{spar} \) (simple paragraph). Every time a paste operation fails, the editor launches a type conversion by searching in the fallback list a possible fallback statement in order to achieve the user’s requirement in spite of the type incompatibility. The algorithm [7] for performing such a conversion checks for the consequences of a conversion for siblings and descendants. If necessary, the conversion of derived changes is automatically triggered.

For one document type made up of \( n \) types, the user has to tediously write at most the \( n^2 - n \) fallback rules. A modular approach of document type definition [22] makes this solution not practical: a document type definition can actually make use of other type definitions, which in turn can make use of other definitions and so on. In one hand, it is cumbersome to put down fallback rules for all these type definitions. On the other hand, the domain and the range of fallbacks seem to be limited to a single type definition and therefore give no possibility of cross—conversions between documents of different types.
6.2 Structured−oriented programming environment

In structure−oriented programming environments, a formal description of languages (a grammar) guides the construction of programs, which are represented as abstract syntax trees. Grammars and abstract syntax trees can be compared with document type definitions and document instance trees. Changes to the grammar description make the programs wrong under the new grammar.

A work [13] based on the Gandalf system [19] tackled this problem and listed the possible elementary transformations, which are the same as those pointed out in this paper. The implementation of the solutions proposed for Gandalf is an environment called TransformGen, in which the changes to the original grammar have to be provided by the user, whereas in our model, the user gives the complete new grammar and changes are automatically detected. At this stage, the main difference is the incapacity of TransformGen for recognizing complex changes such as addition or deletion of structural levels, complex tree restructuring, merging of multiple nodes into a single one or relocating pieces of the tree to new locations.

For these changes, extra action routines may be manually provided; they are performed on the intermediary document resulting from the transformer’s action. An alternative is the manual production of additional rules in a language called ARL [25], that is used by the transformer as complementary resource. This corresponds to the renaming rules described above which, in our case, are not a resource for the Transformer but for the Comparator.

6.3 Object−oriented databases

In the database area, the problem is only partly similar to the one we address in the structured editing systems. The difference is due to the concept of object, which implies, among other notions, inheritance and data abstraction that make the schema update more complicated. Structural inconsistencies concern the static part of the database whereas behavioral inconsistencies refer to the dynamic part of the database that consists in methods. The transformations due to static changes listed in ORION [4] are similar to the elementary transformations presented in this paper. The main difference is the possible violation of the inheritance that doesn’t exist in structured editing systems. Most systems such as O₂ [3], ORION, GemStone [20], ENCORE [23] [24] cover these transformations. Moreover, behavioral inconsistencies, which do not exist in structured editing systems, are not fully covered in database systems. Only O₂ and ENCORE provide mechanisms for partial recovery of these problems.

6.4 Programming languages

The problem of type transformation exists in all typed programming languages and is solved by the coercion mechanism used to change the semantics of operands. It is either statically expressed by the programmer or dynamically detected during run−time type
compatibility check. In most programming languages such as ADA or C, types are assumed equivalent if their names are identical. In other languages such as ALGOL 68, the challenge was to implement structural equivalence rather than name identity. The algorithms were so complex that present languages make shift with name equivalence plus explicit coercion [6].

7 Conclusion

The type model presented in this paper has been applied to some kinds of document transformations. Our analysis leads to the representation of two kinds of trees, namely the type trees and the instance trees. A type tree is a structural representation of a generic type. The main advantage of the structural aspect of types is to consider type transformations by means of structural properties contrary to the name identity formerly used.

We have then built order relations and classes of equivalence based on the notions of structure inclusion (subtype and cluster), structure identity and constructor compatibility. We are therefore quite straightforward about transformations that involve types belonging to the same equivalence class. Moreover, we can infer a relational order over the set of equivalence classes in order to allow transformations between types of different classes, provided a relational order binds such classes together. The transformations handled by the use of these mathematical structures can apply to either static or dynamic transformations.

In order to cope with the dynamic transformations that are not covered by the above structures, a language is associated with each type. We try to satisfy the user’s requirements in a cut and paste operation. It is clear that recovering information is of higher priority than preserving the source structure. Hence, the idea is that a specific logical structure (instance) can be represented by a word which necessary belongs to the language associated with its corresponding generic type. The language helps to point out if the information can be recovered or not. As the same alphabet (basic types) is used in the languages associated to different types, it makes sense to verify if a given word belongs or not to a language of another type, or even to compare languages corresponding to different generic structures.

However, as multiple structures may satisfy an adoption request, two complementary improvements of our model are under investigation: the definition of a cluster dimension and additional information interactively provided by the user.

The document transformation model described in this paper has been implemented in the Grif system, for handling static transformations. Although it performs mainly elementary transformations, this tool has given valuable results, which satisfy practical needs, like transforming automatically documents of type Article into documents of type Chapter. The next step in the development of this tool is to introduce algorithms for type equivalence and type compatibility. Implementation of dynamic transformations based on the language of
types presented in section 5.1 will allow the Grif editor to perform flexible cut and paste operations.

The Grif system is based on the SGML concepts and is therefore well suited to extensions related to DSSSL [18]. The static transformation tool, for instance, is clearly related to the first part of the DSSSL architecture. Our transformation process is an implementation of the DSSSL general language transformation process (GLTP), which provides functions for performing tree transformations. In the DSSSL model, the output definition corresponds to our target generic structure, while the association specification is equivalent to our renaming rules.

In the DSSSL standard, a language called the Location Model has been defined for navigating through hierarchical typed structures. We plan to replace our renaming rule syntax by the Location Model, in order to provide users with a more powerful expression language. However, the DSSSL approach, as presented in the standard is basically static, without any interaction with a user. For dynamic transformations another type of architecture is needed. In fact, this seems to be the main challenge in this research area: how to interactively manage, modify and display highly structured objects.
Bibliography


tecture (ODA)*, (ISO 8613), 1989.


[21] V. Quint and I. Vatton, Grif: an Interactive System for Structured Document Manip-


1986.

Type Modelling for Document Transformation in Structured Editing Systems

1 Document transformation .................................................. 1
   1.1 Structured documents ............................................. 1
   1.2 Needs for document transformation ............................. 2
   1.3 Identifying kinds of structure transformations ............... 3

2 A tree model for document types .................................... 5
   2.1 Document type definition ......................................... 6
   2.2 Tree definition of document types ............................... 7

3 Type definition for static analysis with strong structural constraints .... 10
   3.1 Functions on types ............................................... 10
   3.2 Comparing types .................................................. 12

4 Types equivalence for complex transformations ....................... 14
   4.1 Type isomorphism ................................................ 15
   4.2 Subtypes .......................................................... 17
   4.3 Type clusters ...................................................... 17

5 Tree model for dynamic transformation ................................ 18
   5.1 The language of a type .......................................... 19
   5.2 The paste operation .............................................. 20
   5.3 Application in the Grif editor .................................. 21
   5.4 Multiple solutions for adoption ............................... 22

6 Related work ............................................................. 22
   6.1 Structured editing systems ...................................... 23
   6.2 Structured–oriented programming environment .............. 24
   6.3 Object–oriented databases ..................................... 24
   6.4 Programming languages ......................................... 24

7 Conclusion ................................................................. 25