A Generic Algorithm for Merging SGML/XML-Instances

Design and Implementation for the use case: Function Documentation of Control Unit Software in Automotive Industry

Gerald W. **Manger** <Gerald.Manger@bmw.de>

Abstract

This paper discusses the SGML/XML Generic Merging Algorithm which can be used for merging SGML or XML-files that are valid instances of the same but arbitrary Document Type Definition (DTD). The author outlines that the technical documentation of control unit software motivates a generic merging application. He describes the general requirements for tree-based SGML/XML-instance merging. He gives examples to show how the algorithm works, and discusses some implementation details.

1. A Documentation-System for Control Unit Software

The availability of many data formats is an obstacle for establishing business standards. Industry has an enormous demand for using standard data formats for the exchange of data, which often is made persistent in engineering database systems saved in proprietary formats. SGML and XML might serve as the before mentioned standards.

1.1. A Control Unit Software Development Process

A car has control units for a great variety of functionalities, e.g. motor electronics, safety-functions, heater/air-conditioning-technology, undercarriage control systems etc. For example, a modern motor-management system has lots of motor types (e.g. the number of cylinders, capacity) where software has to be adjusted to several vehicles and miscellaneous external influences. Hence, hundreds of different motor control unit-programs are built, which are the result of a process which consists of the following system development process [W1999]:

1. In the *Requirements Specification* phase, the scope of services for the function which is to be created is specified.

2. In the *Analysis* phase, the system requirements and the system behaviour in relation to its environment are specified by use cases in object models.

3. In the *Function Specification* phase, the control unit functions are specified, which are rather concentrated on physical principles than on implementation details. Control engineering simulator systems model the functions by state charts and block diagrams. The *Implementation* phase, which is included in the Function Specification, contains the target system specific implementation of the developed function.

4. In the *System Configuration* phase, a complete system is built by the available library functions.

5. A complete system, e.g. a motor control unit system, has to be adjusted to a specific variant. In the *Calibration* phase, some thousand parameters have to be calibrated in the control unit to achieve the required motor features. So, a substantial documentation and a system specific configuration of calibration tools is essential.

Each process phase produces its specific data and documents, which are not necessarily known to all involved persons. SGML/XML can serve as the connecting layer (document layer) between the development layer (engineering database system data in proprietary formats) and the presentation layer. Documents have to be exchanged between manufacturer and supplier who often use different tools and data formats. For this reason, a standardized document exchange format is required.

1.2. Industry Standard MSR

The MSR (Manufacturer Supplier Relationship, http://www.msr-wg.de, is an association of several German car manufacturers and electric/electronic suppliers with the purpose of improving the collaboration between manufacturers and suppliers and to increase its efficiency. The technical opportunity of managing all phases of documentation and data exchange between manufacturer and supplier via one format was the decisive factor for choosing SGML/XML as base technology for the

MSR. MSR has developed several DTDs (e.g. for system components, control unit software, network, report) which are used by the concerned companies.

Among other things, the MSRSW.DTD represents the data model for variables and calibration parameters of a control unit. This data is generated in different parallel and sequential process steps. By using the same DTD for all steps and all information fragments in the process, all applications can use the same access paths to the available information. This approach makes the document layer universally applicable.

1.3. Function Documentation eDoc

The BMW Group has the aim of developing an own function documentation process for control unit software based on MSR standard formats, for the reason of serving own documentation purposes and keeping compatible to the MSR DTDs. This documentation process shall be parallel to the software development process, for the reason that there is no delay for the documentation relative to the corresponding software. One further advantage of using these standards is that each company can use the same tools (e.g. formatters) which are commissioned by the MSR.

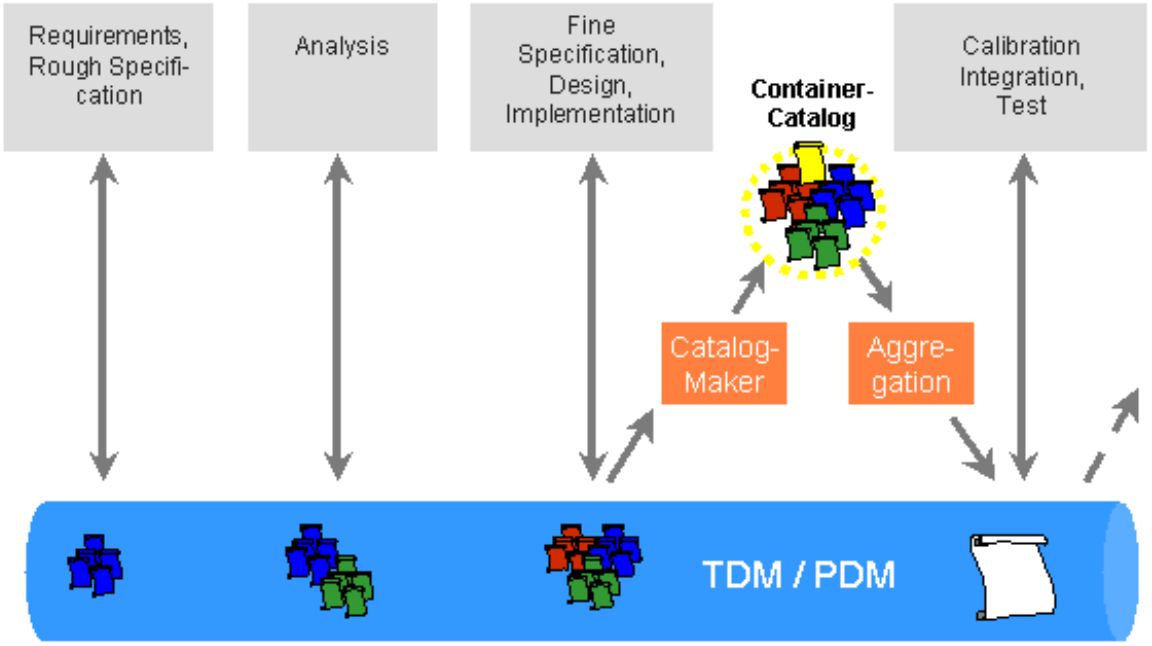

Figure 1. Function Documentation eDoc with MSRSW.DTD-Instances

Figure 1 shows the before explained process phases. The tools behind the phases Requirements Specification, Analysis, (Function-)Specification and Calibration shall generate and also be able to import MSRSW.DTD-instances, which are stored in a SGML/XML-Repository or a Configuration Management System used as Team Data Management (TDM) resp. a Product Management System (PDM).

The System Configuration phase is covered by using the Configuration Management System. The Catalog Maker generates the Container Catalog (CC) from the configuration information. The CC is a SGML/XML-based list of references to documents in several formats and configuration information in the used Configuration Management System.

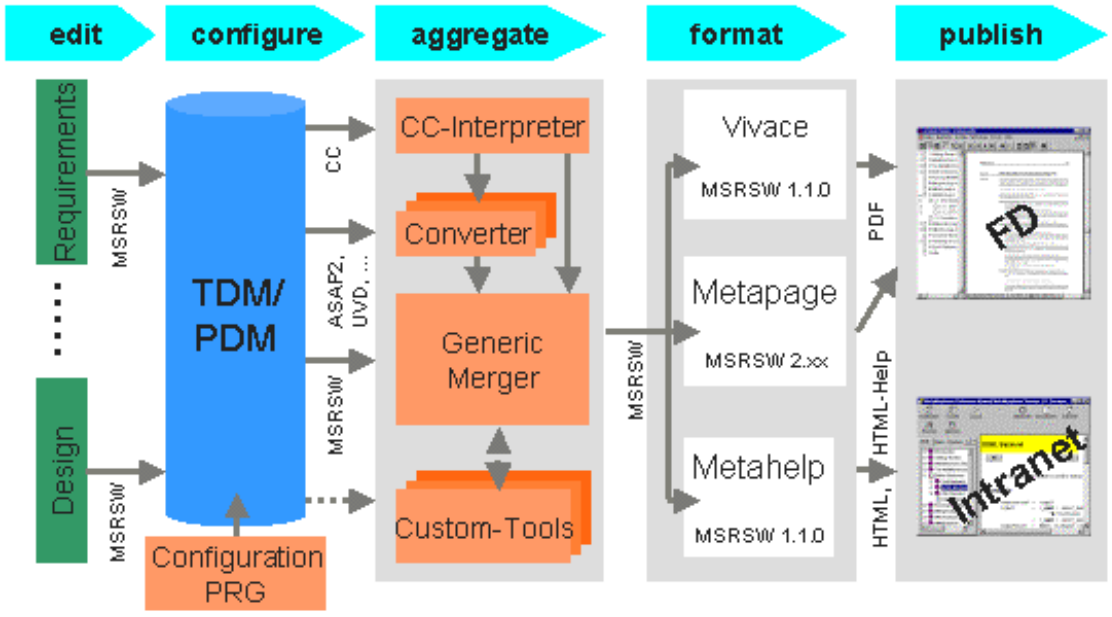

Figure 2. eDoc Modules

The Aggregation of the referenced documents to MSRSW.DTD-instances is illustrated by Figure 2. The dashed line in the lower right corner of Figure 1 indicates the formatted processing of a MSRSW.DTD-instance, which is shown more detailed

in Figure 2.

The MSRSW.DTD-instances in the Configuration Management System / PDM System, the Container Catalog and the referenced files are the input files for the Aggregation. Several converter programs are needed for converting the files stored in proprietary formats to MSRSW.DTD-instances. These converters are called by the Container Catalog Interpreter which takes the CC as input and produces a set of MSRSW.DTD-instances as output. For obtaining a complete document which contains all the MSRSW.DTD-instances corresponding to this set, *a generic merging application for SGML/XML-instances is needed*, for the reason that a DTD can change in its lifecycle.

The complete produced MSRSW.DTD-instance is then processed by a formatter (which is currently Vivace by BOSCH GmbH, and soon Metapage by Ovidius GmbH). Metahelp is commissioned at Ovidius for publishing MSRSW.DTD-instances in the BMW Group Intranet. At the end of the documentation system, the function documentation is stored in the PDM database resp. is published in the BMW Group Intranet. Custom tools have to be considered for future requirements.

Figure 1 and Figure 2 indicate that creating an application for merging SGML/XML-instances is necessary for the documentation process eDoc. The next chapter shows how the SGML/XML-Instances Generic Merging Algorithm, called σ, works [M2000].

2. Definition of Algorithm σ

σ shall merge two or more source instances, which are valid for the same but arbitrary DTD, to a target instance which also is valid concerning the chosen DTD. There are content models which require that one instance is preferred, so a DTD does not allow complete additive merging in general. For this reason, the source instances have to receive a merge priority.

2.1. General Definitions for σ

In this section, some SGML/XML terms are used [G1990]. The following definitions are given:

Δ	set of all DTDs
Φ	set of all valid SGML/XML-instances
Φ^{δ}	set of all valid SGML/XML-instances concerning $\delta \in \Delta$.
E^{δ}	set of all SGML/XML-elements concerning $\delta \in \Delta$
$\varepsilon \in E^{\delta}$	one SGML/XML-element in $\delta \in \Delta$
Γ	set of all SGML/XML-content models
Γ^{δ}	set of all SGML/XML-content models concerning $\delta \in \Delta$
$\Gamma^{\delta}(\varepsilon)$	the content model for element $\varepsilon \in E^{\delta}$, $\delta \in \Delta$

The occurrence indicators of content model-element groups are specified:

*	Any number of instances of an element group is allowed.
+	At least, one instance of an element group has to exist.
?	The element group may be instantiated once or need not to be instantiated.
!	The element group must be instantiated exactly once.

Also, the Γ-element repetition is defined:

Γ-element repetition	Let $\delta \in \Delta$, $\varepsilon \in E^{\delta}$. The content model $\Gamma^{\delta}(\varepsilon)$ has a Γ-element repetition, if an element ε exists twice in $\Gamma^{\delta}(\varepsilon)$. Example: `<!ELEMENT A - - (B,C,B*)>`

σ-modelgroups are different to the modelgroups we know from SGML/XML:

σ-modelgroup	Let $\delta \in \Delta$, $\varepsilon \in E^{\delta}$. A σ-modelgroup of content model $\Gamma^{\delta}(\varepsilon)$ is a highest level element group of $\Gamma^{\delta}(\varepsilon)$, including its occurrence indicator. If the occurrence indicator $\in \{*,+\}$

or if the highest level content model connector is the Or-Connector (in SGML/XML, usually the "|"), $\Gamma^{\delta}(\varepsilon)$ has only one σ-modelgroup (only itself).

$\Gamma^{\delta}{}_{m}(\varepsilon)$ — σ-modelgroup no. m for $\varepsilon \in E^{\delta}$

Example: Let δ = `<!ELEMENT A - - ((B,C),D|E,(F,G)*)>`. The σ-modelgroups are:

1. $\Gamma^{\delta}{}_{1}(A)$ = `(B,C)`
2. $\Gamma^{\delta}{}_{2}(A)$ = `D|E`
3. $\Gamma^{\delta}{}_{3}(A)$ = `(F,G)*`

Example: Let δ = `<!ELEMENT A - - ((B,C)*,D)*>`. In this case, $\Gamma^{\delta}(A) = \Gamma^{\delta}{}_{1}(A)$ = `((B,C)*,D)*`.

A σ-tree node is a tuple (Type,(Attribute,Value)*) in a root tree representation of a SGML/XML-instance with the following allowed types:

`ELEMENT`	the tree node instantiates an element $\varepsilon \in E^{\delta}$, $\delta \in \Delta$
`STRING`	the tree node represents a text element (e.g. `CDATA`)
`PI`	the tree node represents a processing instruction
`COMMENT`	the tree node represents a comment

Example: Let $\delta \in \Delta$ =

```
<!ELEMENT CHAPTER - - ((CHAPTER | (#PCDATA))*)>
<!ATTLIST CHAPTER ID ID #REQUIRED >
```

Let $S \in \Phi^{\delta}$ =

```
<CHAPTER ID="CHP1"> This Chapter consists of ...
<CHAPTER ID="CHP1.1">
<?xm-replace-text {chapter}> This Chapter
describes...</CHAPTER>
```

```
<CHAPTER ID="CHP1.2"> In opposite to Chapter 1.1,
this Chapter describes...</CHAPTER>
</CHAPTER>
```

S is represented by the tree in Figure 3

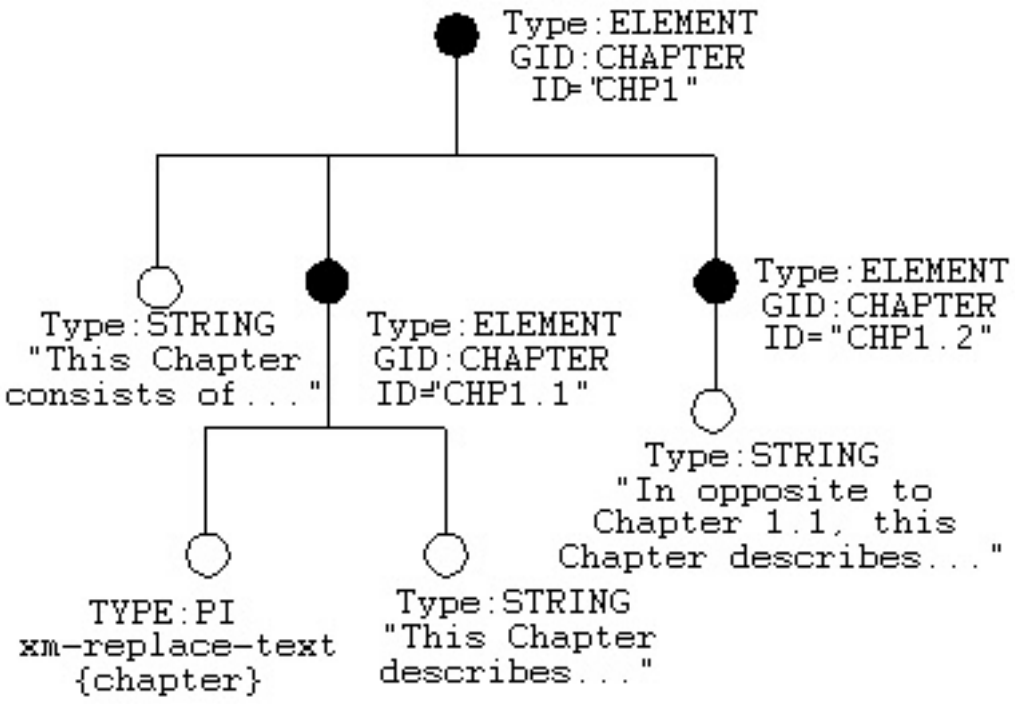

Figure 3. Example of a σ-tree representation

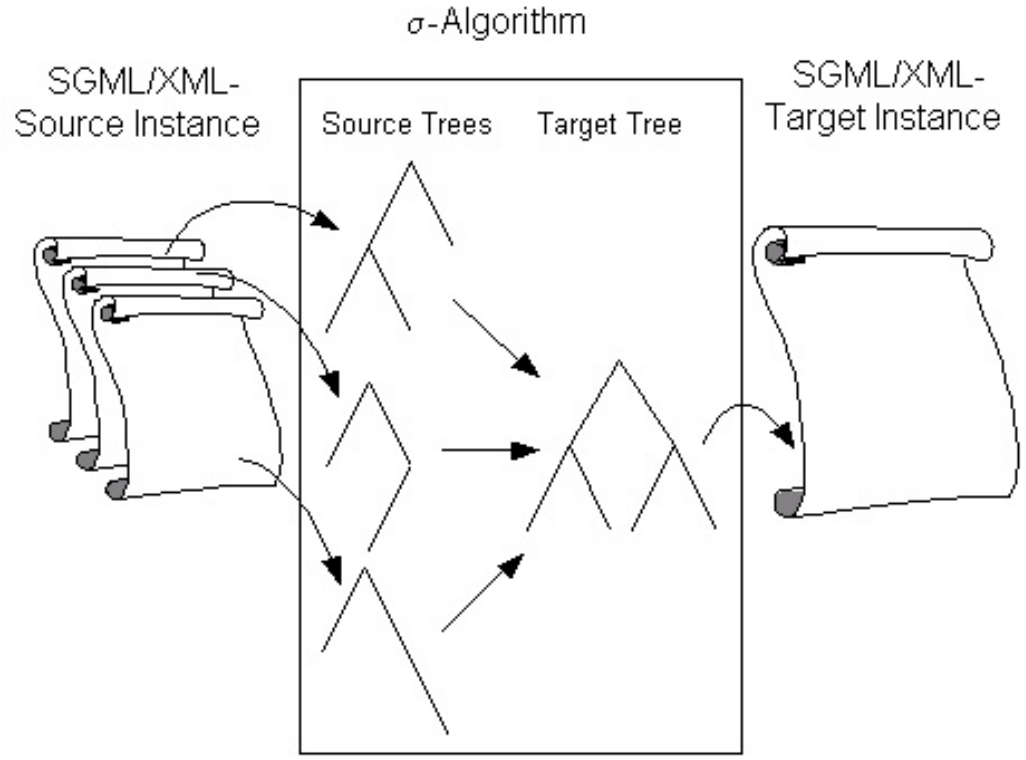

Figure 4. σ-internal representation of source- and target instances

Figure 4 illustrates the components source instances, σ-source trees, σ-target tree, target instance and shows that σ only operates on tree representations of instances.

Since the sequence of instantiated elements of a content model is normally important, the source trees and the target tree have the structure of an ordered root tree [CLR1992]. The following definitions for σ are given, with $\delta \in \Delta$, $S \in \Phi^{\delta}$:

actual σ-target tree node k	this is the actually traversed node of the σ-target tree whose subnodes still have to be added to the σ-target tree
T	set of all ordered root trees
T_{σ}	set of all σ-root trees representing valid SGML/XML-instances
$T^{\delta}{}_{\sigma}$	set of all σ-root trees representing valid

	SGML/XML-instances concerning δ
tr()	Input: a valid SGML/XML-instance $S \in \Phi^{\delta}$
	Output: an ordered σ-root tree in $T^{\delta}{}_{\sigma}$
$K^{\delta}tr(S)$	set of all nodes of tr(S)
root()	Input: a σ-root tree X
	Output: the root node of X
parent()	Input: node $k \in K^{\delta}tr(S)$
	Output: parent node of k, null if k is the root node of tr(S)
type()	Input: node $k \in K^{\delta}tr(S)$
	Output: the node type `ELEMENT`,`STRING`,`PI` or `COMMENT`
gid()	Input: node $k \in K^{\delta}tr(S)$
	Output: $\varepsilon \in E^{\delta}$ if type(k) = `ELEMENT`, else null
Ψ	set of all σ-source trees
Λ	actual σ-target tree during the execution of σ
$K^{\delta}{}_{\Psi}$	set of all nodes of all σ-source trees concerning δ
$K^{\delta}{}_{\Lambda}$	set of all nodes in the actual σ-target tree concerning δ
orig()	Input: node $k \in K^{\delta}{}_{\Lambda}$
	Output: origin node in a σ-source tree from which k is a copy of
occ()	Input: $\Gamma^{\delta}{}_{m}(\varepsilon)$, $\varepsilon \in E^{\delta}$
	Output: the highest level occurrence indicator of $\Gamma^{\delta}{}_{m}(\varepsilon)$
atIDValue()	Input: σ-tree node k_x (source or target tree)

	Output: ID-attribute value if k_x has an ID-attribute, else null
atIDREFValue()	Input: σ-tree node k_x (source or target tree)
	Output: IDREF(S)-attribute value(s) if k_x has an IDREF(S)-attribute, else null
without()	Input: L_1, L_2 as ordered node lists
	Output: ordered node list L_3 with all nodes of L_1 except the nodes of L_2
append()	Input: L_1, L_2 as ordered node lists
	Output: ordered node list L_3 which is constructed by appending L_2 to L_1
$E^{\delta}{}_S$	set of all instantiations in S: tags, textelements, processing instructions or comments in instance S
node()	Input: a tag, a processing instruction, a text element or a comment in $E^{\delta}{}_S$
	Output: the corresponding node in the σ-tree tr(S).
k^c	ordered list of all child nodes of a node $k \in K^{\delta}{}_{\Psi} \cup K^{\delta}{}_{\Lambda}$
$k^c{}_m$	ordered list of all child nodes of a node k concerning $\Gamma^{\delta}{}_m(gid(k))$
$contents_{\Psi}()$	Input: σ-source tree node k_x
	Output: ordered list of all child nodes of k_x
$subtree_{\Psi}()$	Input: σ-source tree node k_x
	Output: ordered σ-subtree of k_x (not including k_x)
σ-path	A σ-path of $k \in K^{\delta}{}_{\Psi} \cup K^{\delta}{}_{\Lambda}$ is the textual concatenation

of gid(root(k)), ..., gid(k); if atIDValue(k_p) is not null for a node k_p lying on the path from root(k) to k, the string [*ID-attribute=ID-attribute value*] is added directly after gid(k_p). The several gid()-values are separated by a slash ("/"). The function σp(k) is the function which calculates a σ-path for a σ-tree node k.

K^{δ} σp(k) — ordered list of all source tree nodes k_i with σp(k) = σp(k_i), sorted ascending by the priority of the corresponding source trees

σ-congruent — Let k_1, $k_2 \in K^{\delta}$ ψ. Two σ-paths σp(k_1), σp(k_2) with σp(k_1) = σp(k_2) are σ-congruent, if the corresponding ancestor nodes of k_1 and k_2 were instantiated by the same mandatory or optional σ-modelgroup which consists of only one element.

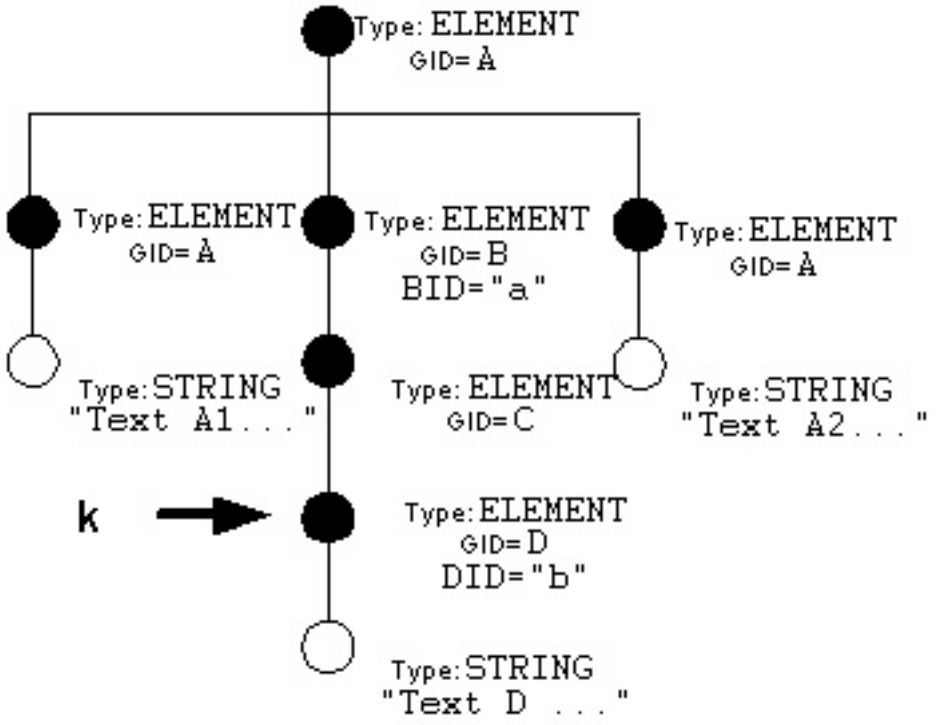

Figure 5. σ-path of a node

The σ-path of node k in Figure 5 is: `"A/B[BID=a]/C/D[DID=b]"`

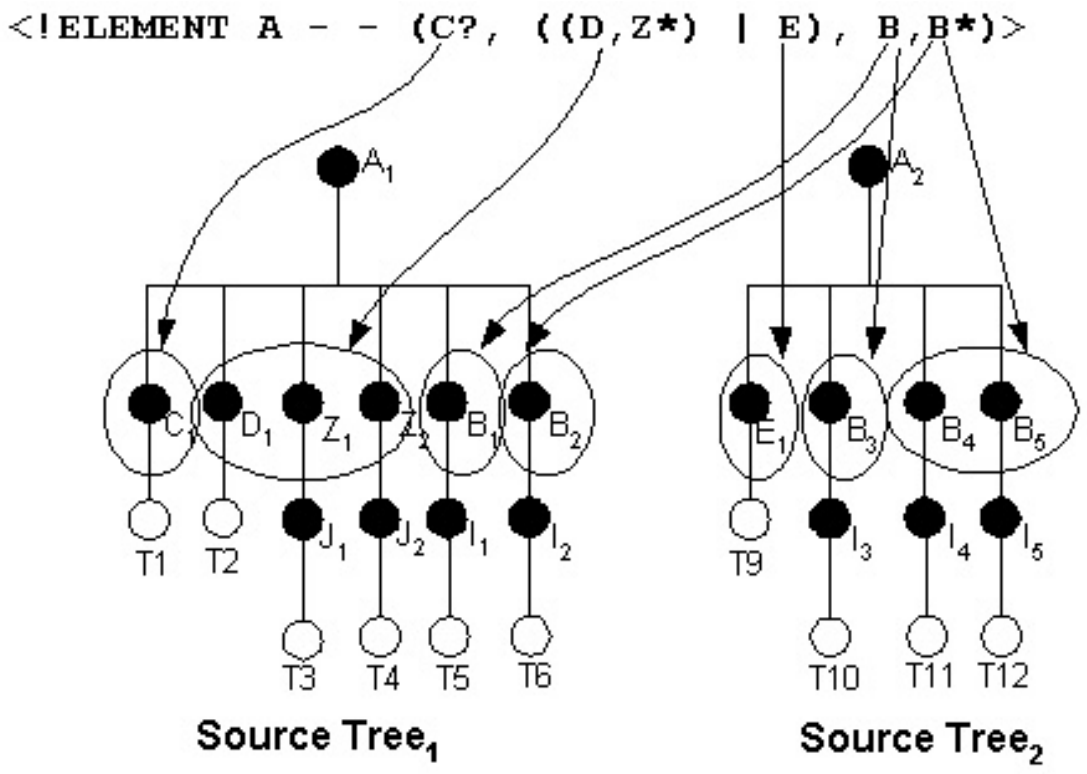

Figure 6. σ-congruent paths

In Figure 6, the paths of node(B_1) and node(B_3) are σ-congruent, since both have the same σ-path `"A/B"` and both were instantiated by the same one-element mandatory σ-modelgroup `B` in the content model of element `A`.

> **Note**
>
> In the following sections, $k \in K^{\delta}{}_{\Lambda}$ denotes the actual σ-target tree node while merging the σ-source trees

2.2. Intuitive Demands for σ

The merging operation σ shall merge σ-source trees representing valid instances concerning the same DTD δ to a σ-target tree representing a valid instance concerning δ. The intuitive demands for such an operation are as follows:

1. The source instances are represented by their corresponding σ-source trees, the σ-target tree is later converted to the target instance.

2. All nodes of the highest priority σ-source tree shall be contained in the σ-target tree.

3. Hence, the σ-target tree has to contain the root node of the highest priority σ-source tree.

4. Only child nodes of type `ELEMENT` which conform to $\Gamma^{\delta}(gid(k))$ may be inserted beneath a σ-target tree node k.

5. Hence, child nodes of a σ-source tree node k_i may only be regarded as potential child nodes of an actual σ-target tree node k if k_i has the same σ-path as k.

6. Since content models may have optional or mandatory σ-modelgroups, only child nodes from *one* σ-source tree may be regarded as child nodes for a σ-target tree node k representing an element having such a σ-modelgroup. One opportunity is to regard only the corresponding child nodes from the σ-source tree with the highest merge priority.

7. Hence, nodes from σ-source trees might be excluded from the σ-target tree. But the child nodes of excluded nodes k_i can be inserted in the σ-target tree if the σ-paths of the nodes k_i are σ-congruent with the σ-path of the actual σ-target tree node k.

8. A target instance with an ID-attribute value existing more than once is not valid, also a target instance with IDREF(S)-attribute values pointing to not existing ID-attribute values.

9. The child nodes k_r of σ-source tree nodes $k_1,k_2,...,k_n$, $\sigma p(k_1) = \sigma p(k_2) = ... = \sigma p(k_n) = \sigma p(k)$, are copied in the σ-target tree beneath node k, grouped by σ-modelgroup and ordered by merge priority, if the corresponding child nodes k_r instantiate the relevant σ-modelgroup.

10. Only the instantiation with the highest available merge priority of a mandatory or optional σ-modelgroup with more than one element is regarded, since as in Figure 6, the first and the second instantiation of σ-modelgroup `((D,Z*) | E)`

are inconsistent.

2.3. getInsertType(k)

A special function is needed which calculates the insert type of an actual σ-target tree node k of type ELEMENT, because it has to be decided whether only orig(k) or all σ-source tree nodes having the same σ-path as k may be regarded as parent nodes for the nodes to be inserted beneath k. The intention behind this is that the complete subtree of orig(k) can be copied to the σ-target tree in the first case.

FUNCTION getInsertType(k)

- Input: $k \in K^{\delta}{}_{\Lambda}$ with type(k) = ELEMENT
- Output: insert_type ∈ {ONLY_FROM_ORIGIN,FROM_ALL_SOURCES}

1. Check the alternatives:
   - IF $| K^{\delta}{}_{\sigma p(k)} |$ = 1 THEN insert_type := ONLY_FROM_ORIGIN
   - ELSEIF Γ^{δ}(gid(parent(k))) has a Γ-element repetition and a node k_i exists in $K^{\delta}{}_{\sigma p(k)}$ with σp(k_i) is σ-congruent with σp(k) THEN insert_type := FROM_ALL_SOURCES
   - ELSEIF k has the sibling nodes k_i with gid(k) = gid(k_i), THEN insert_type := ONLY_FROM_ORIGIN
   - ELSE insert_type := FROM_ALL_SOURCES
2. Return insert_type

The constant ONLY_FROM_ORIGIN indicates that beneath the actual σ-target tree node k, $subtree_{\psi}$(orig(k)) has to be copied. The constant FROM_ALL_SOURCES indicates that the nodes in the list $contents_{\psi}(K^{\delta}{}_{\sigma p(k)})$, depending on the σ-modelgroup, have to be considered for the child nodes of k.

2.4. Algorithm σ without considering ID/IDREF(S)-Attributes

First, both functions are introduced which calculate the nodes to be added beneath an actual σ-target tree node k.

contentsAt() Input: $K^{\delta}_{\sigma p(k)}$, $\Gamma^{\delta}_{m}(gid(k))$

Output: ordered list of nodes in $K^{\delta}_{\sigma p(k)}$ concerning $\Gamma^{\delta}_{m}(gid(k))$

first_ex_contentsAt() Input: $K^{\delta}_{\sigma p(k)}$, $\Gamma^{\delta}_{m}(gid(k))$

Output: first_ex_contentsAt() finds a node k_x in $K^{\delta}_{\sigma p(k)}$ for σ-modelgroup no. m with:

1. k_x is in the σ-source tree with the highest possible merging priority
2. contentsAt((k_x), $\Gamma^{\delta}_{m}(gid(k))) \neq ()$ [1]

and returns contentsAt((k_x),$\Gamma^{\delta}_{m}(gid(k))$). If no node k_x was found with contentsAt((k_x),$\Gamma^{\delta}_{m}(gid(k))) \neq ()$, then () is returned.

Now, the Algorithm σ without considering ID-attributes is defined as follows:

σ without considering ID-attributes

- Input: σ-source trees $tr(S_1),..., tr(S_N)$ with: $\forall k \in K^{\delta}tr(S_i)$, $1 \leq i \leq N$: atIDValue(k) = null, $S_1,...,S_N \in \Phi^{\delta}$, $\delta \in \Delta$.
- Output: σ-target tree Λ.

1. $\Lambda := root(tr(S_N))$
2. k:=root(Λ)
3. Traverse Λ top-down, left-right. The actual σ-target tree node is k.
    1. IF type(k) ∈ {PI,STRING,COMMENT} THEN copy all contents of orig(k)

[1] () denotes the empty list

beneath k and proceed traversing Λ top-down, left-right.

2. ELSE

    1. `insert_type` := getInsertType(k)

    2. IF `insert_type` = `ONLY_FROM_ORIGIN` THEN copy subtree$_{\psi}$(orig(k)) beneath k and stop traversing the subtree of k (but proceed traversing Λ).

       ELSE calculate $K^{\delta}_{\sigma p(k)}$. $\forall\ \Gamma^{\delta}_{m}(gid(k))$ in $\Gamma^{\delta}(gid(k))$:

       - IF $occ(\Gamma^{\delta}_{m}(gid(k))) \in \{*,+\}$ THEN k^{c}_{m} := contentsAt($K^{\delta}_{\sigma p(k)}$,$\Gamma^{\delta}_{m}(gid(k))$)
       - ELSE k^{c}_{m} := first_ex_contentsAt($K^{\delta}_{\sigma p(k)}$,$\Gamma^{\delta}_{m}(gid(k))$)

The following example will show how σ without considering ID-attributes works.

Let δ be the following SGML-DTD:

```
<!ELEMENT A - - (E,B*)>
<!ELEMENT B - - (D)>
<!ELEMENT C - - (F?,(#PCDATA))>
<!ELEMENT (D,F) - - (#PCDATA)>
<!ELEMENT E - - (C*,D*)>
```

S_1, S_2:

```
<A>
 <E>
  <C>
   <F>Text0</F>
   Text1 </C>
  <D>Text2</D>
  <D>Text3</D>
 </E>
 <B>
  <D>Text4</D>
 </B>
</A>
```

```
<A>
 <E>
  <C>Text5</C>
  <C>Text6</C>
 </E>
 <B>
  <D>Text7</D>
 </B>
</A>
```

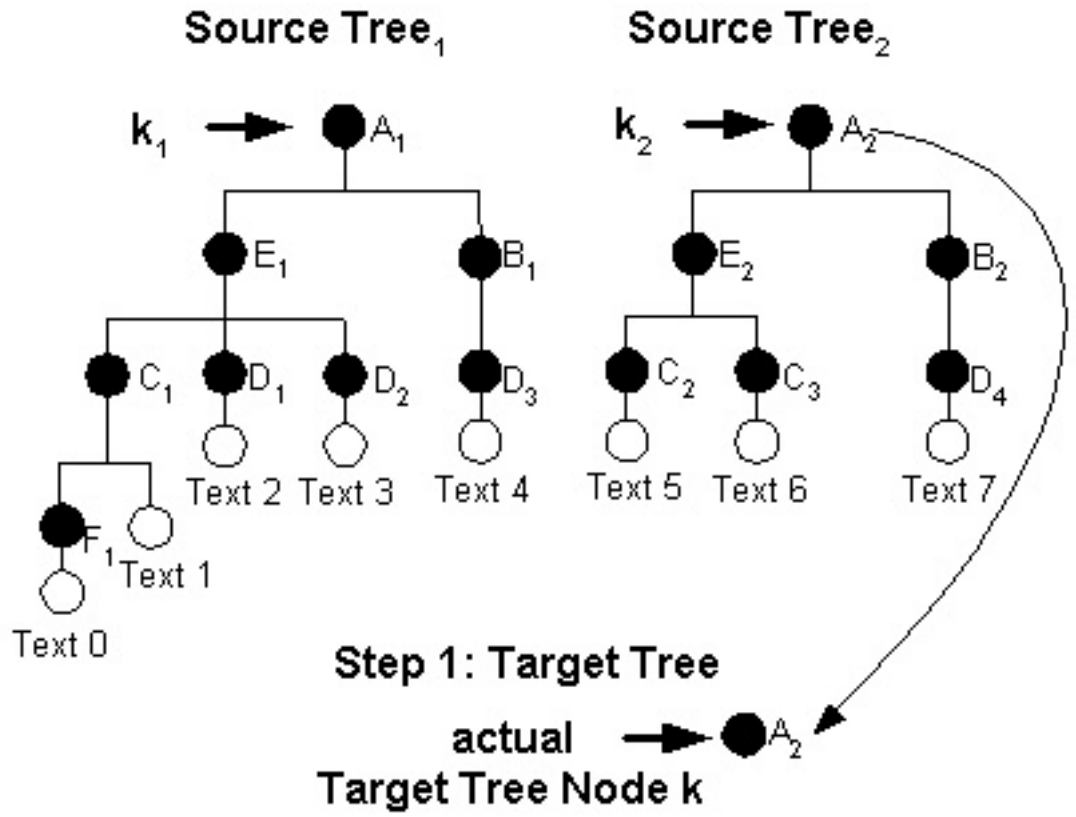

Figure 7. σ-source trees, σ-target tree after the first step

First step: The root of the σ-target tree is set to the root of the σ-source tree with the highest merge priority, this is node k_2 in Figure 7

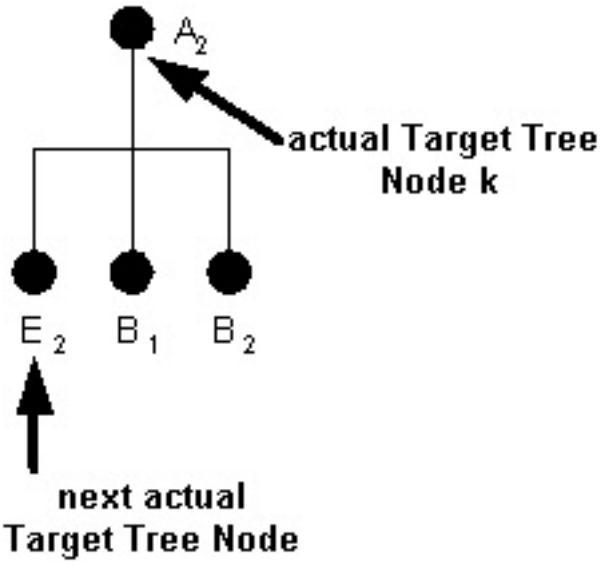

Figure 8. σ-target tree after the second step

Second step: The σ-path of the actual σ-target tree node k is "A". Hence, all σ-source tree nodes with the same σ-path have to be considered, which are k_1 and k2 in Figure 7. Calling getInsertType(k) returns FROM_ALL_SOURCES. The content model for element A which is represented by k has two σ-modelgroups: E with occurrence indicator ! and B* with occurrence indicator *. So both σ-modelgroups have to be handled separately for copying the contents corresponding to the σ-modelgroups beneath k regarding the nodes k_1 and k_2. For σ-modelgroup E, exactly node(E_2) has to be copied, because k_2 is located in a σ-source tree with higher merge priority than the σ-source tree k_1 is located in, and only one E is allowed in the content model of element A. For σ-modelgroup B*, all instantiations beneath k_1 and k_2 are copied beneath k, these are node(B_1) and node(B_2).

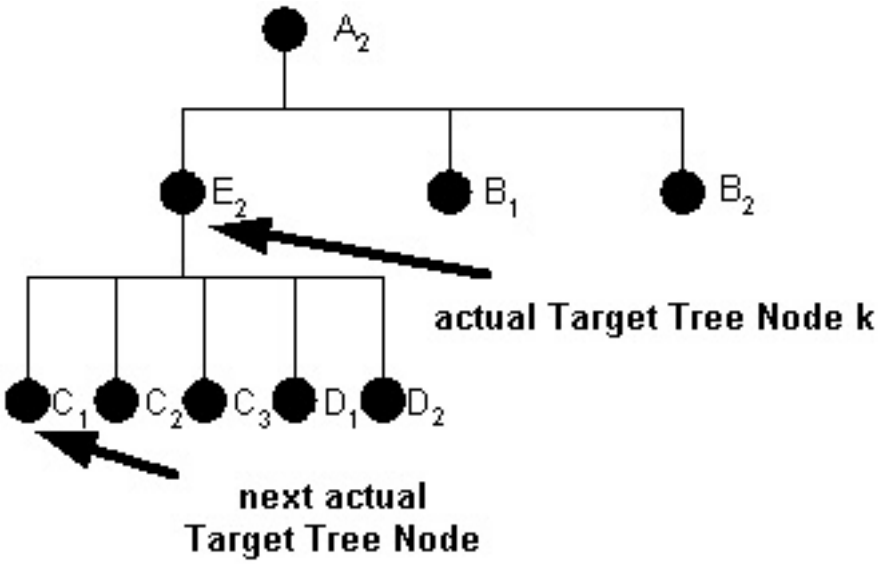

Figure 9. σ-target tree after the third step

Third step: Now, node(E_2) becomes the actual σ-target tree node, because Λ is created and traversed top-down, left right. The σ-path of k is `"A/E"`, so $K^{\delta}_{\sigma p(k)}$ = (node(E_1),node(E_2)). There are two source tree nodes having this path; Γ^{δ}(gid(parent(k))) has no Γ-element repetition and k has no sibling k_x with gid(k) = gid(k_x), so calling getInsertType(k) returns `FROM_ALL_SOURCES`. The content model of the element represented by k has two σ-modelgroups: `C*` and `D*`, these have to be handled separately. For σ-modelgroup `C*`, all instantiations beneath the σ-source tree nodes node(E_1) and node(E_2) are copied, these are the nodes node(C_1), node(C_2) and node(C_3). For σ-modelgroup `D*`, all instantiations beneath the σ-source tree nodes node(E_1) and node(E_2) are copied, these are the nodes node(D1) and node(D_2).

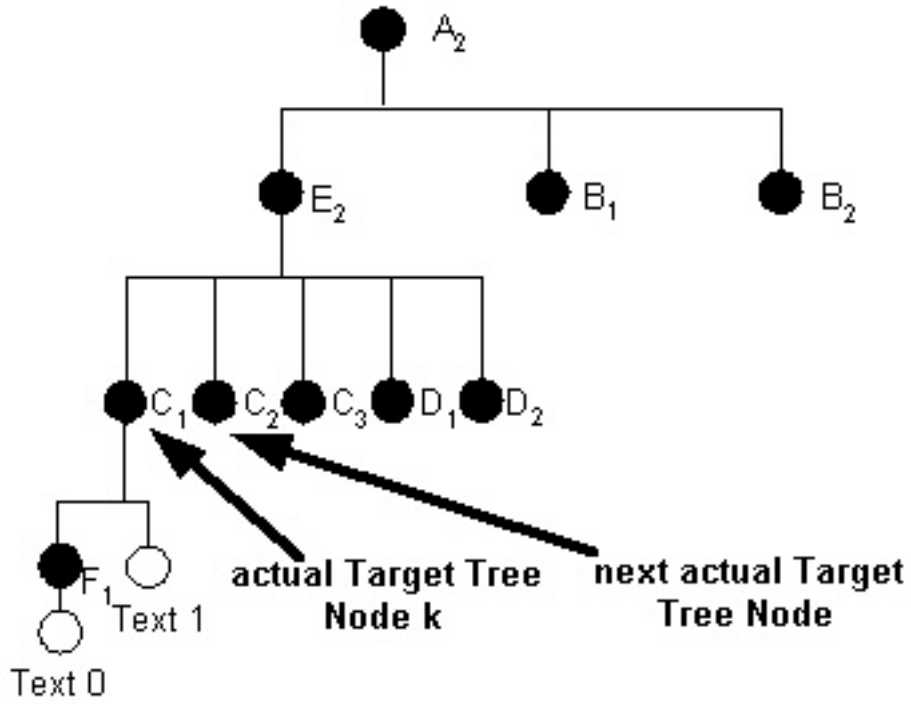

Figure 10. σ-target tree after the fourth step

Fourth step: Now, node(C_1) becomes the actual σ-target tree node k. σp(k) = `"A/E/C"`. Calling getInsertType(k) returns `ONLY_FROM_ORIGIN`, since several σ-source tree nodes exist having this σ-path, but Γ^{δ}(gid(parent(k))) has no Γ-element repetition and k has siblings node(C_2) and node(C_3) which have the same generic identifier `C`. So the complete subtree of orig(k) can be copied beneath k what includes that traversing the descendants of k is not necessary. The resulting tree is shown in Figure 10

The next steps are analogical, until traversing to the σ-target tree node node(D_2). Figure 11 shows the actual σ-target tree after the eighth step.

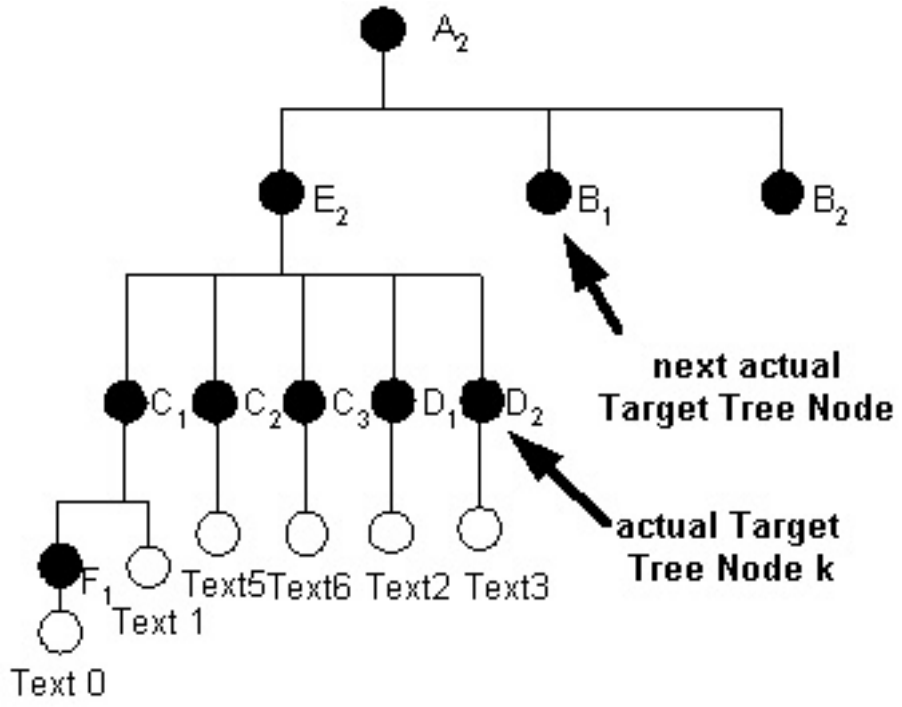

Figure 11. σ-target tree after the eighth step

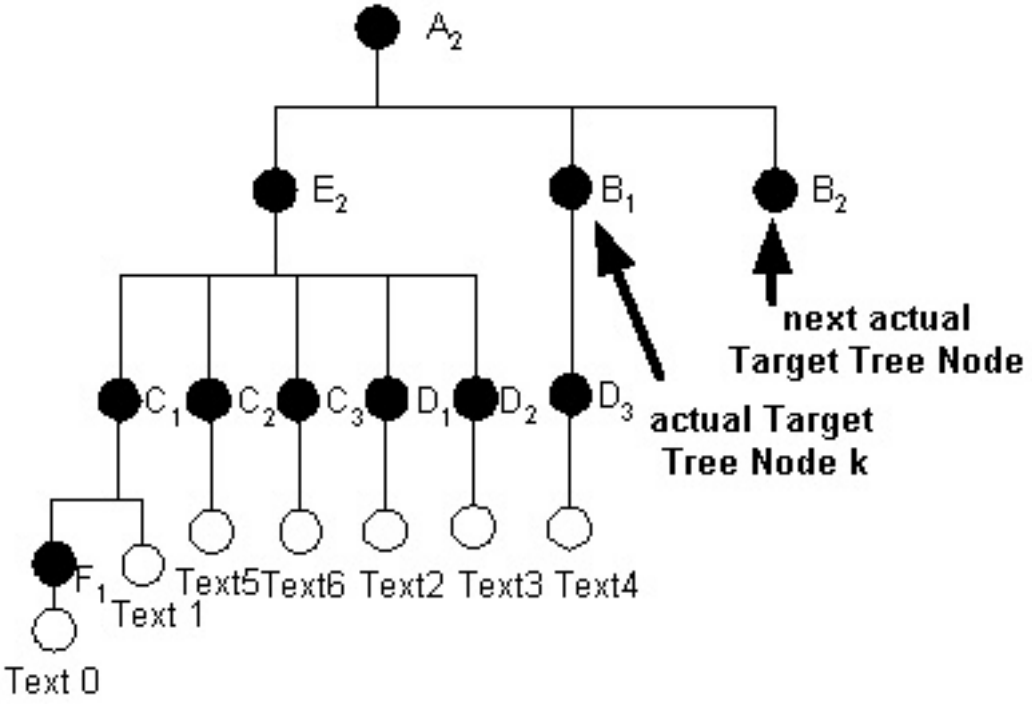

Figure 12. σ-target tree after the ninth step

The last regarded step in this example is node(B_1) as the actual σ-target tree node k. The σ-path of k is "A/B". Γ^{δ}(gid(parent(k))) has no Γ-element repetition and k has siblings with the same generic identifier. Hence, the subtree of orig(k) can be copied beneath k. Figure 12 illustrates the actual σ-target tree.

The next steps are analogical. Figure 13 shows the finished σ-target tree.

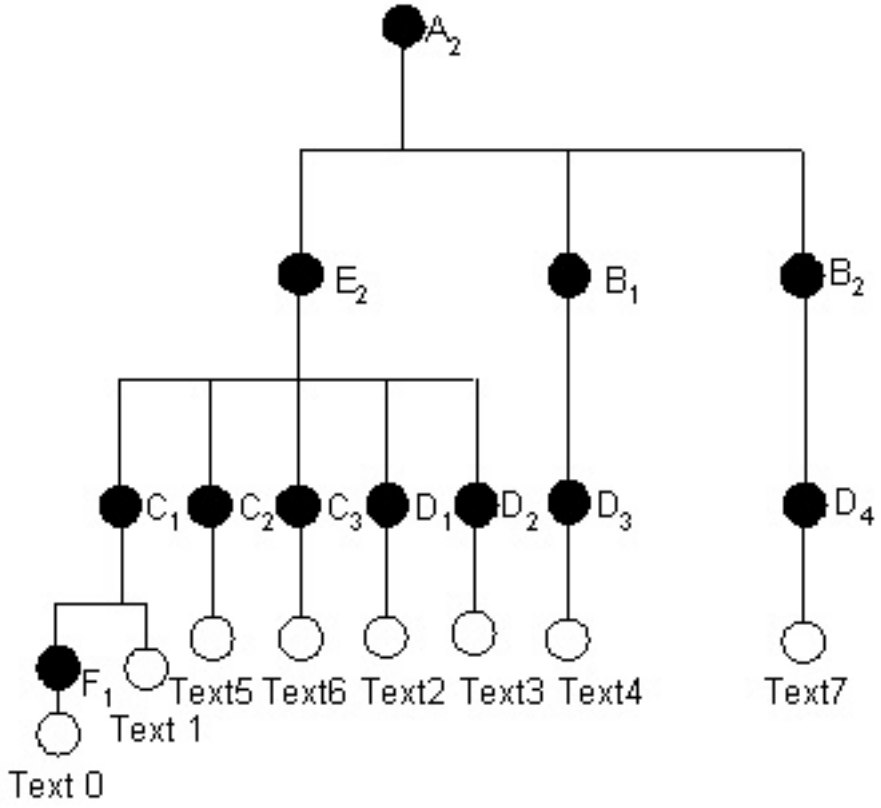

Figure 13. σ-target tree after the last step

2.5. ID/IDREF(S)-Conflicts

For the general use case of σ, now ID/IDREF(S)-attributes have to be covered. These might influence a merging scenario in a way that the resulting instance is invalid concerning the DTD δ. The next definitions specify ID/IDREF(S)-conflicts.

ID-conflict Type 1

Let $\delta \in \Delta$, $N > 1$, $S_1,...,S_N \in \Phi^{\delta}$, Λ the actual σ-target tree, $k \in K^{\delta}{}_{\Lambda}$ the actual σ-target tree node. If nodes $k_x \in$

$K^{\delta}tr(S_i)$, $k_y \in K^{\delta}tr(S_j)$, $i \neq j$, exist with:

1. $atIDValue(k_x) = atIDValue(k_y)$
2. both $parent(k_x)$, $parent(k_y)$ are in $K^{\delta}{}_{\sigma p(k)}$

then we have an ID-conflict of type 1. This is the case when there are σ-source tree nodes having the same σ-path as k, and their child nodes have the same ID-attribute values.

Note

Obviously, an ID-conflict of type 1 is locally detectable.

ID-conflict Type 2

Let $\delta \in \Delta$, $N > 1$, $S_1,...,S_N \in \Phi^{\delta}$, Λ the actual σ-target tree, $k \in K^{\delta}{}_{\Lambda}$ the actual σ-target tree node. If nodes $k_x \in K^{\delta}tr(S_i)$, $k_y \in K^{\delta}tr(S_j)$, $i \neq j$, exist with:

1. $atIDValue(k_x) = atIDValue(k_y)$
2. $parent(k_x)$ is in $K^{\delta}{}_{\sigma p(k)}$, $parent(k_y)$ is not in $K^{\delta}{}_{\sigma p(k)}$

then we have an ID-conflict of type 2. This is the case when there are σ-source tree nodes having the same ID-attribute values, but one cannot handle them locally.

IDREF(S)-conflict

Let Λ be the finished σ-target tree after σ stops. If a node $k_x \in K^{\delta}{}_{\Lambda}$ exists with $y:=atIDREFValue(k_x)$, $y \neq null$, and $\forall k \in K^{\delta}{}_{\Lambda}$: $atIDREFValue(k) \neq y$, then we have an IDREF(S)-conflict. This is the case when an IDREF(S)-attribute's value does not exist as ID-attribute value in the target instance represented by Λ.

2.6. Algorithm σ considering ID/IDREF(S)-Attributes

The algorithm σ is extended in a way that ID-attribute values are involved in solving ID/IDREF(S)-conflicts and in calculating the σ-path.

2.6.1. Solving ID-Conflicts of type 1

If the child nodes of a σ-source tree node k_1 in $K^{\delta}_{\sigma p(k)}$ have an ID-conflict of type 1 with child nodes of another σ-source tree node k_2 in $K^{\delta}_{\sigma p(k)}$, then only those conflict-involved child nodes for all σ-modelgroups $\Gamma^{\delta}(gid(k))$ are copied beneath k which are located in the conflict-involved σ-source tree having the highest possible merging priority. The child nodes of not conflict-involved σ-source trees are also copied concerning to their corresponding σ-modelgroups. The child nodes of the conflict-involved nodes in $K^{\delta}_{\sigma p(k)}$ are not copied.

Algorithm for handling ID-conflicts of type 1

- Input: actual σ-target tree node k.
- Output: ordered list of all unusable nodes $U^{\delta}_{\sigma p(k)}$ in $K^{\delta}_{\sigma p(k)}$, having child nodes being conflict-involved in an ID-conflict of type 1 but not being of the highest merging priority.

1. $U^{\delta}_{\sigma p(k)} := ()$
2. For all pairs (k_a, k_b), k_a, k_b in $K^{\delta}_{\sigma p(k)}$, $k_a \in K^{\delta}tr(S_s)$, $k_b \in K^{\delta}tr(S_t)$, $s < t$, having child nodes $k_i \in contents_{\psi}(k_a)$, $k_j \in contents_{\psi}(k_b)$:

   - IF $atIDValue(k_i) = atIDValue(k_j)$ /* ID-conflict of type 1 */

     THEN $U^{\delta}_{\sigma p(k)} := append(U^{\delta}_{\sigma p(k)},(k_a))$

Node k_a is appended to the unusable nodes list because k_a has a lower priority than k_b, since $s < t$. After $U^{\delta}_{\sigma p(k)}$ is calculated, the nodes to be added beneath k are calculated for each σ-modelgroup:

$\forall\ \Gamma^{\delta}_{m}(gid(k)) \in \Gamma^{\delta}(gid(k))$:

- $k^{c}_{m} := contentsAt(without(K^{\delta}_{\sigma p(k)},U^{\delta}_{\sigma p(k)}),\Gamma^{\delta}_{m}(gid(k)))$

Note

The child nodes of the unusable conflict-involved nodes can still be regarded for the merging scenario in the next step, if they have the appropriate σ-path.

The following example shows how an ID-conflict of type 1 is handled and illustrates that the child nodes of unusable conflict-involved nodes are copied in the next target tree level. Given the SGML-DTD δ:

```
<!ELEMENT A - - (B,C,D)>
<!ELEMENT B - - (E)*>
<!ELEMENT (C,D,F) - - (#PCDATA)>
<!ELEMENT E - - (F)*>
<!ATTLIST E ID ID #REQUIRED>
```

Given $S_1, S_2, S_3 \in \Phi^{\delta}$:

```
<A>
 <B>
  <E ID="A1">
   <F>Text1</F>
   <F>Text2</F>
  </E>
 </B>
 <C>Text3</C>
 <D>Text4</D>
</A>
```

```
 <A>
  <B>
   <E ID="A2">
    <F>Text5</F>
    <F>Text6</F>
   </E>
  </B>
  <C>Text7</C>
  <D>Text8</D>
 </A>
```

```
 <A>
  <B>
   <E ID="A2">
    <F>Text9</F>
    <F>Text10</F>
   </E>
  </B>
  <C>Text11</C>
  <D>Text12</D>
```

```
</A>
```

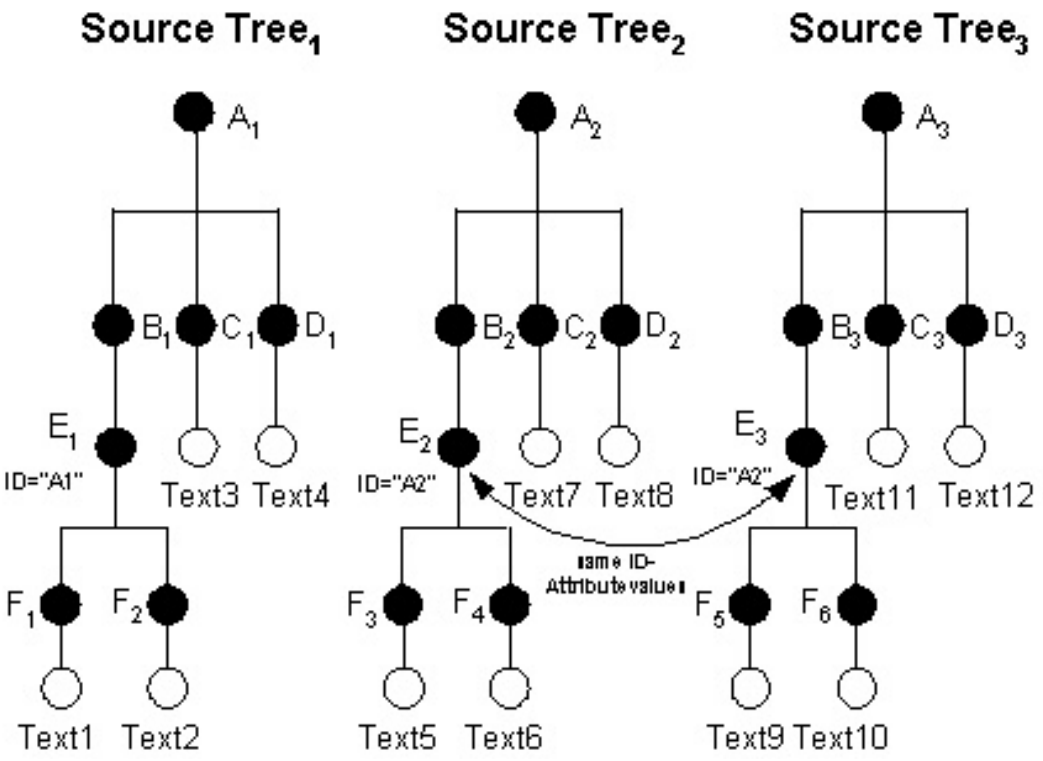

Figure 14. Strategy for ID-conflict of type 1

The σ-source trees for S_1, S_2, S_3 are shown in Figure 14. First, node(A_3) is copied as root node in the σ-target tree. Since element `A` has only the mandatory σ-modelgroups `C`, `D` and `E` in δ, only the instantiations of the highest merge priority σ-source tree are taken: node(B_3), node(C_3) and node(D_3). Traversing top-down, left right results in node(B_3) getting the actual σ-target tree node k. Three σ-source tree nodes have the same σ-path `"A/B"` as k: node(B_1), node(B_2) and node(B_3). Hence, the child nodes of these three nodes have to be checked on ID-conflicts. We have an ID-conflict of type 1 here, because the σ-source tree nodes node(E_2) and node(E_3) have the same ID-attribute value `"A2"`. This results in copying the conflict free node(E_1) and the conflict-involved and higher merge priority node(E_3). If node(E3) becomes the actual σ-target tree node k, all σ-source tree nodes are calculated which have the same σ-path `"A/B/E[ID=A2]"`. These nodes are node(E2) and node(E_3). The content model of element `E` is `(F)*`, so all child nodes

$_2$) and node(E3) are copied. The σ-target tree is shown in Figure 15.

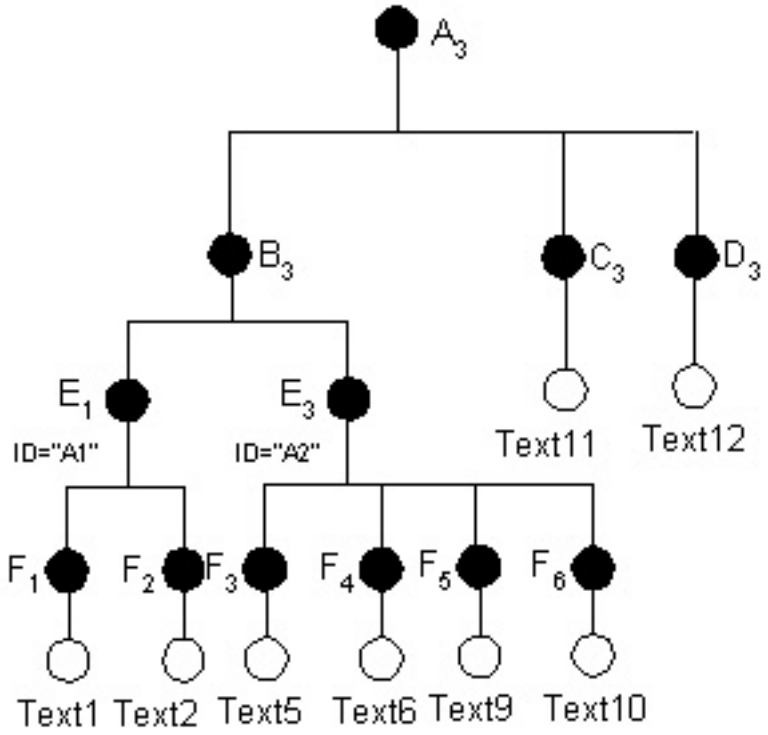

Figure 15. σ-target tree after solving ID-conflict of type 1

2.6.2. Detecting ID-conflicts of type 2

An ID-conflict of type 2 is detected if the actual σ-target tree node k has an ID-attribute value which already exists in the set of all ID-attribute values of all σ-target tree nodes. Obviously, an ID-conflict of type 2 is not locally detectable. Hence, the σ-merging operation has to be cancelled if two σ-target tree nodes with the same ID-attribute values exist.

> **Note**
>
> An ID-conflict of type 2 does not automatically imply an invalid target instance!

2.6.3. Handling IDREF(S)-conflicts

As long as the σ-target tree traversion is not finished, it is not possible to predict in a generic way if a node will be inserted which has a special ID-attribute value

referenced by a σ-target tree node having an IDREF(S)-attribute value. So, after the merging operation has finished, one can validate the resulting instance in order to look for invalid IDREF(S)-attribute values.

3. Implementation of σ

SIGMA (SGML/XML-Instances Generic Merging Application) as the implementation of σ was coded in MetaMorphosis, which is a declarative, rule-based programming language used to transform one structure into another structure, e.g. XML into HTML or a comma separated value-file into SGML using a certain grammar [O2001]. Since σ is generic, it is important to have a DTD-Plugin which allowes to access the actual used DTD and validate temporary tree structures.

MetaMorphosis has two standard-rules: the `!begin`-rule and the `!default`-rule. The `!begin`-rule is executed at the beginning of a transformation and normally copies the source tree root-node (or source tree root nodes, if more than one input-source is given) in the target structure. The `!default`-rule is executed for all nodes during a transformation which were not assigned a certain rule. If no explicit `!default`-rule is given, the child nodes of the actual σ-target tree node are copied. SIGMA changes the standard-implementation of the two rules. The `!default`-rule was coded in SIGMA so that the content model of the actual σ-target tree node can be read and an appropriate action can be executed, as shown in the previous sections. The `!begin`-rule is coded in two separate `!begin`-rules. The `!begin !down`-rule is for the first visit of the target structure where the highest priority σ-source tree node has to be copied and a parallel structure for a content model tree has to be maintained. The `!begin !up`-rule for the last visit of the target structure is where temporary structures have to be removed and some preparations for the SGML or XML-output have to be done. Also important is the opportunity to stop the traversation of a certain subtree and copy it completely into the σ-target tree by the `!copy !continue`-directives.

SIGMA is configured by a certain SGML-instance. This allowes the user to specify

- the files to be merged

- to choose the conflicts to be checked
- to decide whether the target instance is to be validated
- merging priority changes for nodes with a special σ-path
- elements as semantic identifiers influencing the σ-path instead of an ID-attribute value
- the occurrence indicator of an arbitrary content model

SIGMA does not support the following optional SGML-features, which are not supported by XML either: `CONCUR`, `DATATAG`, `IMPLICIT`, `EXPLICIT`, `RANK`, `SIMPLE`, `SUBDOC`.

Bibliography

[CLR1992] Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., 1992.*Introduction to Algorithms*, Eighth printing, The MIT Press, Cambridge, pp.86-94.

[G1990] Goldfarb, Charles F.,1990.*The SGML Handbook*, Clarendon Press, Oxford.

[M2000] Manger, Gerald W., 2000. *Ein generischer Algorithmus zur Verschmelzung von SGML/XML-Instanzen*, Diploma-Thesis at the Johann Wolfgang Goethe-University Frankfurt/M. in cooperation with the BMW Group Munich, Frankfurt/M., Germany.

[O2001] Ovidius GmbH, 2001. *MetaMorphosis Reference Manual*, http://www.ovidius.com/download/mmrefman-pdf.zip

[W1999] Weichel, Bernhard, 1999. *SGML/XML als Verbindungsschicht in Entwicklungsprozessen*, Robert Bosch GmbH, Schwieberdingen, Germany.

Glossary

CC	Container Catalog

PDM	Product Management System
TDM	Team Data Management

Biography

Gerald W. **Manger**
Development Engineer
BMW Group
Integration of Technical Data, Development Documentation
Munich
Germany
Email: Gerald.Manger@bmw.de

Gerald W. Manger - Gerald W. Manger is Development Engineer at the Research and Engineering Center of the BMW Group in Munich/Germany. He develops methods using SGML/XML-technologies for the technical documentation of control unit software for motor management systems. Before, he finished his studies of Computer Sciences with the Diploma at the Johann Wolfgang Goethe-University in Frankfurt on the Main/Germany.