Sketchy Modeling for XML

Jüri Kiho

University of Tartu, Estonia,
kiho@cs.ut.ee,
http://www.ut.ee/~kiho

Abstract. General principles of sketchy modeling techniques are described. XML is used for defining the modeling language. In turn, a particular sketchy model for XML is developed.

1 Introduction

The main domain of sketchy modeling (SKM) techniques lays at the edge of human-computer interaction – at the touching point where data units of "common interest" must be clearly "understandable" both for human beings and computers. Such data form a huge application area, including all kinds of computer texts – program source texts as well as any other textual user-written input data or user-read output data. The sketchy modeling method offers a uniform and, at the same time, highly flexible framework for graphical representing and handling computer texts' structures.
In 1998, a uniform sketchy text model was proposed [1]. The perfected model, suitable for any kind of structured computer texts, was specified in 2000 [2]. The model and general modeling principles are briefly explained in present paper (Sections 2 and 3). By that, XML is used to describe the sketchy modeling language (SKML). A project (*AmadeusJ*) is under development, which aims to develop an integrated environment for handling sketchy texts, i.e. computer texts modeled in accord with the sketchy text model. Both multi-view and multi-base features are supported: graphical views as well as base languages are considered as variable attributes assigned to certain parts of text structures. The system is open, allowing easily to add new views and base languages. Currently, it supports designing Java programs, LaTeX documents and HTML texts. The main features of *AmadeusJ* are characterized in [2].
In this paper (Section 4), principles for sketchy modeling of XML texts are developed. Integrating this model into the sketchy processing system *AmadeusJ* enables to design and handle XML documents in any graphical view, supported by the system. Works on introducing sketchy XML have already been started. The first versions of the relevant tools, such as sketchifier, textualizer, reducer and exporter have been added to *AmadeusJ*.

2 The Sketchy Text Model

2.1 General

The sketchy text model provides a uniform technique for constructing complex computer text structures, using elementary text units as atomic building blocks. Any correspondingly structured text is called the *sketchy text*. The elementary text units are supposed to have the row-structure, close to that for usual plain texts.

A *text unit* is a sequence of rows together with a special row, called the *comment* (of the text unit). A *row*, in turn, is a sequence of row elements.

In general, the *row element* is either a *symbol* – <character, font, color>, or a hyper-symbol [2]. However, for our purposes here, it is sufficient to be restricted only to "plain" symbols: we shall consider any row as a sequence of ASCII characters.

Any sketchy text is obtained by arranging text units (and other attributes) into a tree-like structure, called the sketchy tree.

2.2 The sketchy tree

The *sketchy tree* is a pair $<T, A>$ where A is a set of arrows and T is a non-empty tree with the following properties.

There are three kinds of nodes: *sketch* (node of kind "sketch"), *branch* (node of kind "branch"), and *primitive* (node of kind "primitive").

The root of T is a sketch. All successors of a sketch are branches, successors of a branch can be of kind "primitive" or "sketch". Primitives have no successors.

We will assume natural level-numbering, defining the value $v.level$ for node v as follows:

if v is root, $v.level = 0$, otherwise $v.level =$ (predecessor of v)$.level + 1$.

It follows, that branches occur only at odd levels.

In addition, some "backward" arrows may be given (in set A), describing certain "back-threading" in the sketchy tree.

Set A consists of zero or more pairs $<p, i>$ (called the *arrows*), where p ("start" of the arrow) is a primitive node in T and $0 \leq i < p.level$ and

$$< p, i > \in A \wedge < p', i' > \in A \wedge < p, i > \neq < p', i' > \Rightarrow p \neq p'.$$

The last condition means that only one arrow can "start" from a primitive.

It is assumed that arrow $<p, i>$ "points" to the node at level i in the path from root to p.

Any sketchy tree can be considered as a hierarchical set of nested structures. According to such viewpoint, a sample sketchy tree is drawn in Figure 1. This particular sketchy tree contains three sketches (s1, s2, s3), six branches (b1, ..., b6), eight primitives (p1, ..., p8) and four arrows (<p2, 0>, <p4, 0>, <p7, 1>, <p8, 2>). Such a representation style is called the *sketchy view*.

The sketchy view appears to be one of the most convenient views for modeled common computer texts, allowing to depict and arrange sketchy tree attributes (e. g. text units and comments) in a very reasonable manner.

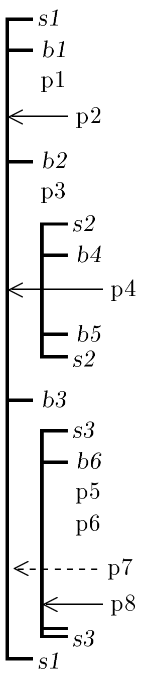

Fig. 1. A sketchy tree as a hierarchy of sketches and branches. A sketch is shown by a (prolonged) left bracket enfolding all its successor-branches. A branch starts with a separator "—" and contains the (vertically located) sequence of its successors. Solid arrows "point" to sketches. Dotted arrows "point" to branches.

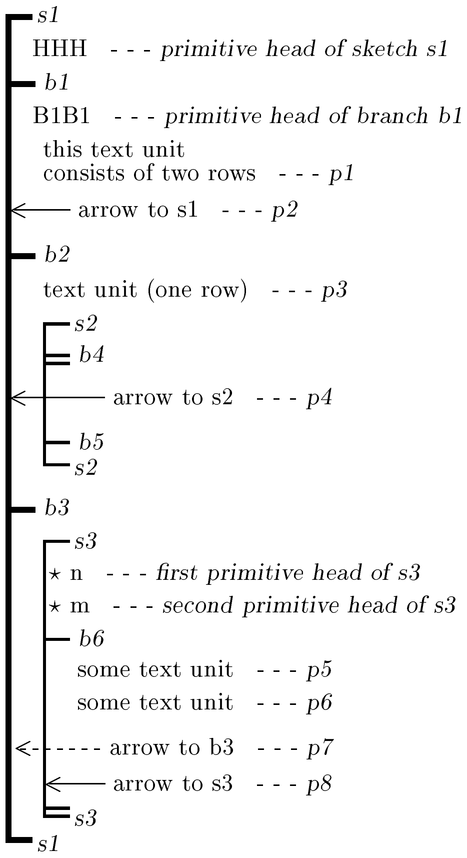

Fig. 2. A sketchy text as an attributed sketchy tree

Table 1. Reflection of scheme types in the sketchy print style

Scheme	Type	Reflected in figure as
Sketch	0	left bracket
Sketch	1	left bracket with double end-mark
Sketch	2	boldface left bracket
Branch	0	separator — (single)
Branch	1	separator = (double)
Branch	2	separator **—** (single, bold)

2.3 Head attributes

The term "scheme" will be used as a common name for notions "sketch" and "branch". Every scheme is assigned a *head* (the head attribute) – a sequence of *primitive heads*. If the sequence is empty then the scheme is *headless*. The attributes for primitive heads will be introduced further on.

2.4 Textual attributes

The textual attributes are:

– scheme comments – attributes for sketches and branches
– text units – attributes for primitive nodes and primitive heads

Every scheme is assigned a row (see Section 2.1), called the *scheme comment*. Every primitive head is assigned a text unit. Every primitive (i.e primitive node) is assigned a text unit. In Figure 2, a sketchy text obtained by attributing the sketchy tree from Figure 1 is depicted.

Note. Throughout present paper, for drawing sketchy texts, we use the sketchy print style, supported by *AmadeusJ* . In this style, italic is used for comments, and, in addition, text unit comments are prefixed by "- - -". The sketch comment is always duplicated at the end of the sketch bracket. The primitive heads are depicted at the beginning of schemes. The type attributes (see next section) are reflected too.

2.5 Type attributes

Every node in a sketchy tree and every primitive head is assigned a *type* – a non-negative integer. A sketch (branch) of type 0 is called the *simple sketch* (*simple branch*). A sketch (branch) of type 1 is called the *loop sketch* (*loop branch*). A sketch (branch) of type 2 is called the *module sketch* (*module branch*).
In Table 1 the sketchy print style drawing conventions for some scheme types are given. So, the sample sketchy text in Figure 2 is a sketch (s1) of type 2 containing three branches (b1, b2, b3) of type 2. Sub-sketch s2 is of type 0, its

branches (b4, b5) are of types 1 and 0 correspondingly. Sub-sketch s3 is of type 1 and contains a branch (b6) of type 0. Non-zero types of primitive nodes and primitive heads are shown by special prefixes. In particular, primitive head type 1 reflects as "$\star$" (see Figure 2, the primitive heads of sketch s3). The full set of primitive type prefixes for the sketchy view styles, supported by *AmadeusJ*, is given in [2]

2.6 Other attributes

The rest of attributes

- icon-tags
- views
- base languages

are for sketches and branches.
Every scheme is assigned an *icon-tag* – a boolean value. If icon-tag is *true*, the scheme is displayed as an icon (a small filled rectangle in the sketchy print style). In Figure 3, the sample sketchy text from Figure 2 with *true* icon-tags for branches b1 and b2 is depicted.
Every scheme is assigned a *view*, which defines its display style – screen planning and drawing for this particular scheme. A view defines a concrete display style for corresponding scheme. More precisely, the view, assigned to scheme s, controls composing the screen picture for that particular scheme, given its desirable location. If the icon-tag of s is *true*, one of the trivial predefined small pictures (an icon) is used. Otherwise, a structured screen picture is composed and displayed, which depicts the scheme as a collection of screen pictures of all constituents of s together with complementary graphics (lines, boxes, etc.).
Types of schemes and primitives are usually shown by means of complementary graphics.
It should be noted that schemes in one sketchy text may have different views. Therefore parts of one sketchy text can be displayed in different styles.
Text units have no own views, they are displayed vertically, row by row. Only the location of the text unit comment may vary from view to view. Any row is displayed horizontally, element by element.
Every scheme is assigned a *base language*, which defines all semantics-sensitive operations applicable for the scheme. The base language attribute may be empty. It means, that no semantics-sensitive operation is available. Normally, it is assumed that text units, assigned to primitive heads of schemes and primitive nodes of branches, are written in corresponding base language. As a rule, the base language attribute is not directly reflected in any view.

2.7 The modeling language

In Figure 4, the syntax of sketchy texts is given, using XML document type definition (DTD). It does not reflect the constraints imposed on arrows' endpoint levels. Element type TEXT is used for *text unit*.

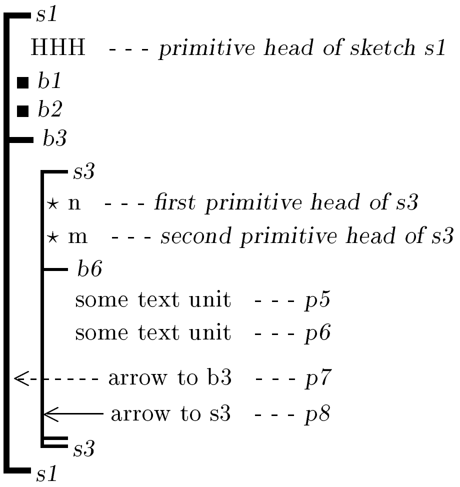

Fig. 3. A sketchy text with icon-tags set for branches b1 and b2

3 Modeling

Sketchy modeling of computer texts is a redesigning activity, which aims to restructure plain computer texts to the form of sketchy texts. Provided that there exist tools for handling sketchy texts, any modeled source text can be graphically edited as well as viewed in an appropriate display style.
In a broader sense, sketchy modeling means representing any kind of data in the form of sketchy texts, i. e. the sketchy form may be the only one, "understandable" both for human beings and computers.
Let us consider a certain data domain together with the language L providing rules for representing any particular data collection (source data) in this domain. The sketchy model for language L consists of exact rules, defining which construct in L corresponds to which construct in the sketchy text model. In other words, the sketchy model defines a *mapping* from L to sketchy text language. The mapping must be reversible – for any sketchy text based on this model, its origin in L must be constructively defined.
For instance, in the case of a common programming language, the designer should, in particular, decide which simpler constructs (e. g. declarations, expressions, assignment operators, calls) form text units and how to represent more complex constructs (e. g. modules, control structures) using sketches, branches, arrows, primitives and primitive heads of certain type. Sketchy modeling may be useful for a large variety of data domains. It is most suitable for inherently hierarchical, repeatedly nested data or text structures.
The following general design recommendations for defining sketchy mappings should be kept in mind.
For sketchy texts, obtained from a plain-text base language, any well-defined base language construct should be allowed as a primitive text unit, even if a

```
<!ELEMENT SKETCHY_TEXT (SKETCH) >
<!ELEMENT SKETCH
        (COMMENT?, PRIMITIVE_HEAD*, BRANCH+) >
<!ELEMENT BRANCH
        (COMMENT?, PRIMITIVE_HEAD*, (PRIMITIVE|ARROW|SKETCH)*) >
<!ELEMENT PRIMITIVE_HEAD (TEXT) >
<!ELEMENT PRIMITIVE (TEXT) >
<!ELEMENT ARROW (TEXT) >
<!ELEMENT TEXT (COMMENT?, ROW*) >
<!ELEMENT ROW (#PCDATA) >
<!ELEMENT COMMENT (#PCDATA) >

<!ENTITY % schemeTypes "(0|1|2)" >
<!ENTITY % schemeHeadTypes "(0|1|2|3|4|5|6|7)" >
<!ENTITY % primitiveTypes "(0|1)" >
<!ENTITY % arrowTypes "(2|3)" >
<!ENTITY % views "(0|1|2|3|4|6|7|8|9)" >
<!ENTITY % baseLanguages "(0|1|2|3|4)" >
<!ENTITY % schemeAttributes
        'type %schemeTypes; "0"
        icon (yes|no) "no"
        view %views; "0"
        base %baseLanguages; "3" '>

<!ATTLIST SKETCH %schemeAttributes; >
<!ATTLIST BRANCH %schemeAttributes; >
<!ATTLIST PRIMITIVE_HEAD type %schemeHeadTypes; "0" >
<!ATTLIST PRIMITIVE type %primitiveTypes; "0" >
<!ATTLIST ARROW type %arrowTypes; "2" level CDATA #REQUIRED >
```

Fig. 4. DTD for SKML

sketchy representation for such construct has been designed.
For any kind of sketchy texts, some extra grouping facilities should be allowed. The user should have the possibility to enclose any part of his/her sketchy text into an extra sketch (of specific type, for instance), which only purpose is grouping. Such extra groupings must be irrelevant to the semantics of the sketchy text, and ignored by any semantics-related processing. It gives the user much higher flexibility for additional structuring larger sketchy texts, first of all – for iconifying ("hiding", or "shrinking") any currently non-actual part.

4 Modeling XML Documents

In this section, the rules for mapping plain XML documents to sketchy texts, i.e. sketchy XML documents, are given.
Any sketchy XML document is described in the first branch of an outer sketch. The types of the outer sketch and the first branch, as well as their other elements (comment, head, additional branches) are insignificant. For better structuring, bigger XML constructs, such as DTD (and constituents), elements, comments, CDATA sections etc. may be represented as sub-sketches (instead of plain-text markup). The base language for all schemes is XML, therefore the attribute *base* has value "4" for every scheme. Everywhere in a sketchy XML document, non-commented simple sketches (with arbitrary heads) may be freely used as the extra grouping facilities.
Throughout this section, we shall define every particular mapping as an overloaded function $SK()$ which takes a source construction as its argument, and yields the corresponding sketchy text construct as its value. To obtain necessary strictness, the value of $SK()$ will be specified as XML element, conforming to DTD in Figure 4. In this model, arrows are not involved at all – the sketchy XML model is arrow-less.
In general, character data portions from XML elements are mapped to primitives, and attribute=value pairs are mapped to primitive heads. The row-structure of the source character data portions and attribute=value pairs is preserved as much as possible: the text units included into primitives and primitive heads are split into rows in accord to occurrences of newline characters in corresponding source strings. For instance, the character data portion, say $string =$

```
Some source text in
two rows.
```

will be mapped (sketchified) to a text unit $SK(string)$ of the following structure:

```
<TEXT>
      <ROW> Some source text in </ROW>
      <ROW> two rows. </ROW>
</TEXT>
```

Such mapping of strings (character data, attribute=value pairs et al.) is rather obvious from the example, so we skip more precise definition of $SK(String)$.

4.1 Modeling XML Elements

Any XML element is mapped as a simple sketch containing one simple branch. The element type is assigned to the sketch comment and its attribute=value pairs are mapped as primitive heads of the sketch (see Figure 5). More precisely, mapping of XML elements is defined as follows.
Let E be an XML element, $E =$

<$elementType \quad attributesPart$>
$content$
</$elementType$ >

Then $SK(E) =$

<SKETCH base="4">
<BRANCH base="4">
<COMMENT>
$elementType$
</COMMENT>
$SK(attributesPart)$
$SK(content)$
</BRANCH>
</SKETCH>

Let A be an attributes' part of some XML element, $A = avp_1\, avp_2\, \ldots\, avp_n$, where avp_i $(i = 1, 2, \ldots, n)$ is an attribute=value pair (string), $n \geq 0$; $n = 0$, if A is empty. Then $SK(A)$ is a sequence of n members, where the i-th member $(i = 1, 2, \ldots, n)$ is:

<PRIMITIVE_HEAD>
$SK(avp_i)$
</PRIMITIVE_HEAD>

Note, that the function $SK()$, applied to string argument (avp_i) yields an XML element of type TEXT as described in the previous section.
Let C be a content of some XML element,
$C = cu_1\, cu_2\, \ldots\, cu_n$,
where $cu_i (i = 1, 2, \ldots, n)$ is a content unit, i.e. character data (string) or entity reference (string) or CDATA section or processing instruction or comment or (nested) element. Then $SK(C)$ is a sequence of n members, where the i-th member $(i = 1, 2, \ldots, n)$ is:

$SK(cu_i)$

Any CDATA section is mapped as a simple sketch containing one module branch. The last has one primitive head of type 0, containing text "CDATA" (see Figure 6).
More precisely, let CD be a CDATA section, $CD =$ <![CDATA[$cdString$]]>

a) plain:

```
<BOOK title="XML"
      rank="5">
       ... content ...
</BOOK>
```

b) sketchified:

BOOK
title="XML"
rank="5"

SK(... *content* ...)
BOOK

Fig. 5. A sample element's model

a) plain:

```
<![CDATA[
nothing <= any-
thing]]>
```

b) sketchified:

CDATA
nothing <= any-
thing

Fig. 6. A sample CDATA section

a) plain:

```
<!-- No
comments! -->
```

b) sketchified:

No
comments!

Fig. 7. A sample comment

a) plain:

```
<!DOCTYPE SET
      SYSTEM "set.dtd"
      [ internal subset ] >
```

b) sketchified:

!! SET
SYSTEM "set.dtd"

SK(*internal subset*)

Fig. 8. A DTD for SET

a) plain:

```
<![%switch;[
      <!ENTITY BODY
            "HELP, SOURCE">
]]>
```

b) sketchified:

%switch;

? BODY
"HELP, SOURCE"

Fig. 9. A sample conditional section

Then $SK(CD) =$

<SKETCH base="4">
    <BRANCH type = "2" base="4" >
        <PRIMITIVE_HEAD> $SK(\text{"}CDATA\text{"})$ </PRIMITIVE_HEAD>
        <PRIMITIVE> $SK(cdString)$ </PRIMITIVE>
    </BRANCH>
</SKETCH>

Any XML comment is mapped as a loop sketch containing one simple branch. The latter has a single primitive of type 0, containing the comment text (see Figure 7).

More precisely, let $Comm$ be a comment, $Comm =$ <!-- $commentText$ -->

Then $SK(Comm) =$

<SKETCH type = "1" base="4">
    <BRANCH base="4" >
        <PRIMITIVE> $SK(commentText)$ </PRIMITIVE>
    </BRANCH>
</SKETCH>

A processing instruction anywhere in the sketchy XML document is modeled as a primitive of type 1, containing the processing instruction's text. (In the sketchy print style, primitive of type 1 is suffixed by "?", see Figure 10.)

4.2 Modeling XML DTD

A DTD in some XML document is modeled as a module sketch with head. The first primitive head must be of type 5. (In the sketchy print style, such primitive head is prefixed by "!!".) The text(s) of the primitive head(s) contain the name of the root element and, possibly, the external DTD subset reference (see Figure 8). Declarations inside DTD for element types, entities, attribute lists and notations are represented by module branches. The kind of a declaration is specified by the module's primitive head type.

Let D be a declaration in a DTD, $D =$ <!α $nameSpec$ $valueSpec$>

where $\alpha \in \{$ELEMENT, ENTITY, ATTLIST, NOTATION$\}$; $nameSpec$ and $valueSpec$ are considered as strings (in particular, the first one starts with % and some white space(s) in the case of parameter entity declaration).

Then $SK(D) =$

<BRANCH type="2" base="4">
    <PRIMITIVE_HEAD type="β">
        $SK(nameSpec)$
    </PRIMITIVE_HEAD>
    <PRIMITIVE>
        $SK(valueSpec)$
    </PRIMITIVE>
</BRANCH>

where $\beta = 6$ if $\alpha =$ELEMENT, $\beta = 3$ if $\alpha =$ENTITY, $\beta = 1$ if $\alpha =$ATTLIST, $\beta = 2$ if $\alpha =$NOTATION. (In the sketchy print style, the primitive heads of types 6, 3, 1, 2 are prefixed by tags "! ", "?", "$\star$", "??" correspondingly.)

a) plain:

```
<?xml version='1.0' standalone='yes' ?>
<!-- Extracted from [3], file: Parts.xml -->
<?xml-stylesheet type="text/css" href="Inventory01.css"?>
<INVENTORY>
      <BOOK>
            <TITLE>The Adventures of Huckleberry Finn</TITLE>
            <AUTHOR>Mark Twain</AUTHOR>
            <PAGES>298</PAGES>
      </BOOK>
      <BOOK>
            <TITLE>Leaves of Grass</TITLE>
            <AUTHOR>Walt Whitman</AUTHOR>
            <PAGES>462</PAGES>
      </BOOK>
</INVENTORY>
```

b) sketchified:

xml version='1.0' standalone='yes' **?**

Extracted from [3], file: Parts.xml

xml-stylesheet type="text/css" href="Inventory01.css" **?**

INVENTORY
BOOK
TITLE
The Adventures of Huckleberry Finn
TITLE
AUTHOR
Mark Twain
AUTHOR
PAGES
298
PAGES
BOOK
BOOK
TITLE
Leaves of Grass
TITLE
AUTHOR
Walt Whitman
AUTHOR
PAGES
462
PAGES
BOOK
INVENTORY

Fig. 10. A sample XML document

SKML DTD. The external subset

ELEMENTS

! SKETCHY_TEXT
(SKETCH)

! SKETCH - - - *attributes: type, icon, view, base*
(COMMENT?, PRIMITIVE_HEAD*, BRANCH+)

! BRANCH - - - *attributes: type, icon, view, base*
(COMMENT?, PRIMITIVE_HEAD*, (PRIMITIVE|ARROW|SKETCH)*)

! PRIMITIVE_HEAD - - - *attribute: type*
(TEXT)

! PRIMITIVE - - - *attribute: type*
(TEXT)

! ARROW - - - *attributes: type, level*
(TEXT)

! TEXT
(COMMENT?, ROW*)

! ROW
(#PCDATA)

! COMMENT
(#PCDATA)

■ *ENTITIES (see Figure 12)*

■ *ATTLISTS (see Figure 13)*

Fig. 11. The sketchified DTD for SKML

ENTITIES

? % schemeTypes
"(0|1|2)"

? % schemeHeadTypes
"(0|1|2|3|4|5|6|7)"

? % primitiveTypes
"(0|1)"

? % arrowTypes
"(2|3)"

? % views
"(0|1|2|3|4|6|7|8|9)"

? % baseLanguages
"(0|1|2|3|4)"

? % schemeAttributes
'type %schemeTypes; "0" - - - *simple*
icon (yes|no) "no" - - - *not iconified*
view %views; "0" - - - *sketchy*
base %baseLanguages; "3" ' - - - *none*

Fig. 12. The ENTITIES part

ATTLISTS

⋆ SKETCH
%schemeAttributes;

⋆ BRANCH
%schemeAttributes;

⋆ PRIMITIVE_HEAD
type %schemeHeadTypes; "0"

⋆ PRIMITIVE
type %primitiveTypes; "0"

⋆ ARROW
type %arrowTypes; "2"
level CDATA #REQUIRED

Fig. 13. The ATTLISTS part

4.3 Modeling DTD External Subset

An external DTD subset is modeled as a simple sketch. Similarly to internal subset, declarations inside the external subset for element types, entities, attribute lists and notations are represented by module branches. In Figures 11 – 13, the sketchy representation of the sketchy text DTD (see Figure 4) is depicted.
In addition, any conditional section is mapped as a module branch, which has one primitive head of type 0, containing text "INCLUDE" or "IGNORE" or relevant entity reference (see Figure 9). More precisely, let Cs be a conditional section, $Cs =$ <![$inclueOrIgnore$[$CsContent$]]>
Then $SK(Cs) =$

<BRANCH type = "2" base="4" >
    <PRIMITIVE_HEAD> $SK(inclueOrIgnore)$ </PRIMITIVE_HEAD>
    <PRIMITIVE> $SK(CsContent)$ </PRIMITIVE>
</BRANCH>

5 Conclusion and Further Work

It has been demonstrated how XML could be used to describe SKML and, vice verse, a sketchy model for XML has been defined. The model serves as a basis for introducing the new base language – XML – into *AmadeusJ*.
It would be of certain interest to investigate formal relationships between XML and SKML. Compared to XML, SKML reduces the burden upon the attribute mechanism, since it has a number of "built-in" element attributes and explicit, though restricted, means (arrows) for defining some inter-element references.
Also, the further development of sketchy XML is necessary. The present model, based upon the sketchifying function $SK()$, should be properly widen to a sketchy XML design language. The most popular "sub-languages" i.e. processing instructions languages (such as Javascript and Perl) should be modeled and incorporated into *AmadeusJ* too. Obviously, such kind of (bigger) processing instructions would be reasonable to model by separate sub-sketches, each having the "sub-language" as its base attribute. A special new view might be added to *AmadeusJ*, where at least the standard sketchy view prefixes (e.g. "!!", "!", "?", "*") would be replaced by corresponding XML keywords (e.g. DOCTYPE, ELEMENT, ENTITY, ATTLIST).

References

1. Kiho, J., Hagu, V., Sutt, M.: A text model for the Java-based model-view multieditor. Proc. *Nordic Workshop on Programming Environment Research NWPER'98* , June 14-16, 1998, Bergen, Informatics report No 152, Information science report No 66, 221–225
2. Kiho, J.: SKM. Sketchy Modeling of Computer Texts. *Research Report, Jan. 2000*, Inst. of Computer Science, Univ. of Tartu
3. Young, M., J.: XML Step by Step. *Microsoft Press* 2000