Designing Computer Texts with *AMADEUS* ⋆

Jüri Kiho ⋆⋆ , Roman Ustitch ⋆⋆⋆ , Mart Tudre †

University of Tartu, Institute of Computer Science §

Abstract. *AMADEUS* is a multi-view multi-language general-purpose system for handling computer texts, such as program sources, *HTML* documents, *LATEX* texts et al. It supports graphic representations of the computer texts and their editing "in the large", beside the usual text editing means. The editing functions are independent of the nature of the current source, provided that the latter is mapped as so-called sketchy text. The sketchy text model offers rather general opportunities for reasonable text structuring. In this paper, particular methods for mapping *Java*, *HTML* and *LATEX* texts are introduced. On their basis, designing corresponding source texts in a graphic environment may be carried out effectively and most conveniently.

1 Basic concepts

Despite of powerful graphic abilities of nowadays computers, the most common tools used to represent structure of computer source texts are still those coming from the era of alphanumeric input/output devices, namely tabbing, spacing, and new-lining. The project *AMADEUS* [1] aims to develop a *Java*-based multi-language multi-view general-purpose graphic text editor. To some extent, its functionality is comparable to *EMACS*, but the main syntax constructs are displayed graphically and handled as higher level units of the base (sketchy) text model. There are also tools for mapping existing plain source texts onto the model (providing structural views of them), as well as tools to convert modeled (sketchy) texts to corresponding plain source texts.

1.1 Sketchy text model

Any of the computer text considered in *AMADEUS* is called the *sketchy text* and is based on the *sketchy text model* [2]. In this model, some smaller text fragments (called the *primitives*) are grouped into larger constructs called the *schemes*. A scheme consists of its type (a number), a comment (text row), a head (null or

⋆ Partly supported by Estonian Science Foundation, grant no 3343

⋆⋆ Assoc. professor; kiho@csd.ut.ee; +372 7 375474

⋆⋆⋆ Undergraduate student; romanu@csd.ut.ee

† Graduate student; tudre@csd.ut.ee

§ J. Liivi str. 2, Tartu, 50409, Estonia; +372 7 375445

more primitives), a body. There are two kinds of schemes: sketches and branches. The sketch body consists of branches, while the branch body consists of primitives and sketches. Each primitive is also assigned its type (a number). Any sketchy text is represented as a sketch.
The sketchy text model in *BNF* is formally described as follows.

———————————————————————————

Meta-symbols:
{ } – "0 or more"
/ – "or"
Predefined symbols: *number*, *character*, *font*, *color*, *image*.
scheme ::= sketch / branch
sketch ::= scheme_type scheme_head branch{branch} scheme_comment
branch ::= scheme_type scheme_head {member} scheme_comment
scheme_type ::= *number*
scheme_head ::= {primitive_head}
primitive_head ::= head_type text
head_type ::= *number*
scheme_comment ::= row
member ::= sketch / primitive_member
primitive_member ::= primitive / arrow
primitive ::= primitive_type text
arrow ::= arrow_type endpoint_level text
arrow_type ::= *number*
endpoint_level ::= *number*
text ::= {row} text_comment
text_comment ::= row
row ::= {row_element}
row_element ::= symbol / *image*
symbol ::= *character font color*
sketchy_text ::= sketch

———————————————————————————-

Although the scheme types are just numbers, three of them (0, 1, and 2) are currently supported, and schemes of these types are called *simple*, *loop*, and *module* correspondingly. A plain text in some computer language is represented also as a sketchy text, called the *plain sketchy text*. In such case, the latter is a single-branch simple scheme with single-row text primitives.

1.2 Views

Every scheme is assigned a view (class) from the set of possible ones. The view is used to display the scheme by its methods for screen planning and drawing. The figures in this paper are drawn in the *sketchy print-style*, which is very close to the display view-style called *ViewSketchy*. The main difference between these two sketchy styles is the manner of specifying scheme elements (comments, heads etc). While the display view uses different colors for the elements, the print style uses italic for comments and, in addition, the primitive comments are prefixed by "- - -". In both views, the sketch comment is always shown also at the end of the sketch line. Branches are attached to their parent-sketch lines (and have no own "branch lines"). To specify branch type, corresponding type-tag is shown in front of branch comment: "–" for simple branch, "=" for loop branch, and "—" for module branch. The prefix is omitted in the case of a single simple branch with empty comment.

1.3 Base languages

Every scheme is assigned a base language (class) from the set of possible ones. The base language class contains specific methods for transforming the scheme, including sketchifying (converting current plain sketchy text into structured sketchy text), textualizing (converting current sketch into plain sketchy text), and reducing (simplifying the structure of the current sketch).

2 Main features of the multi-editor

The multi-editor *AMADEUS* handles sketchy texts. Beside usual text editing tools (for primitives), it supports also editing tools to handle (insert/delete, toggle types etc.) the schemes and primitives as a whole. In particular, any scheme may be easily selected (by mouse-left-click) and iconified/deiconified (by mouse-right-click). It also allows to sketchify, textualize, or reduce the selected sketch in the active window using corresponding methods provided by the base language of the selected sketch. Sketchifying and textualizing result in a new sketchy text shown in a new window, but can be performed "in place" too. Reducing performs only "in place".

It enables to choose a view for the selected scheme, or set a view for it and for all of its sub-schemes. It enables to inspect the base language (type) of the selected scheme, or set a base language for it and for all of its sub-schemes. It has export/import functions, using brows-able HTML-files for storing sketchy texts. Read/write functions transfer between plain source texts and plain sketchy texts.

3 Designing computer texts

AMADEUS currently supports three base languages: *Java*, *LATEX*, *HTML*. To add a new one, the new base language class inherited from the abstract class *BaseLanguage* must be written, implementing corresponding methods (*sketchify*, *textualize*, *reduce*). As a basis, the mapping of texts (written in this base language) onto the sketchy text model must be carefully designed and defined. After including the new class into the system, *AMADEUS* is ready to be used for designing and developing computer texts in this base language too.
In the following, some possibilities to map computer text onto the sketchy text model are considered. Beside obtaining well-structured texts, the redundant typing of various end-tags is excluded.

3.1 Mapping for Java

The sketchy representation was originally developed for program source texts [3]. The semantics of primitive conditions and arrows to represent various control structures inside methods is still the same (see figure 1). Any *Java*-class is mapped as a module sketch with head (figure 2). Its body contains one branch, corresponding to the class body. The head contains the text preceding the class body. A *try-catch* statement is mapped as a simple sketch with an empty head of specific type. The first one corresponds to the body of the *try*-clause, the next branches correspond to the *catch*-clauses.

```
BinarySearch
static int BinarySearch(item a[ ],
item x, int left, int right)
throws ItemNotFoundException

display(a, left, right);
(left == right) ?
(x.equals(a[left])) ?
left;
throw new
ItemNotFoundException();
int mid = (int)((left + right) / 2);
(x.equals(a[mid])) ?
mid;
(x.less(a[mid])) ?
BinarySearch(a, x, left, mid);
BinarySearch(a, x, mid+1, right);
BinarySearch
```

Fig. 1. Method *BinarySearch*

```
This sketchy text is generated by AMADEUS
import javagently.Text;
import java.io.*;
class binary
class binary  - - -  (the origin in [4])

■ Javadoc
■ BinarySearch (Figure 1)
main
public static void main(String[ ]
args) throws IOException

. . .
- - - try several searches:
..while
..try
!!

. . .
item x = new Item(
Text.readChar(in));
..try
!!

int found =
BinarySearch(a, x, 0,
a.length-1);
. . .
..catch
(ItemNotFoundException
! e)
. . .
..try
..catch
! (EOFException e)
;
..try
..while
main
display
static void display(item[] a,
int left, int right)

. . .
display
class binary
■ class ItemNotFoundException
■ class item
This sketchy text is generated by AMADEUS
```

Fig. 2. Sketchy Java source

3.2 Mapping for HTML

Below the principles for the sketchy representation of *HTML* documents are explained, using just print-outs form *AMADEUS* help files (see figures 3 – 8). These files themselves are also structured sketchy texts. The little black boxes correspond to iconified sub-schemes, their content is shown in separate figures. Brackets '[' and ']' are used as meta-symbols to denote optional elements.

HTML mapping

Any sketchy HTML text is a sketch, containing a simple sketch with comment "HTML" as the single member of its first branch.

The type of the outer sketch, as well as its other elements(comment, head(s), additional branches) are insignificant.

For better structuring, tagged elements of the sketchy HTML text may be represented as simple subschemes (instead of plain-text tagging). Loop schemes can be used for block comments, and module schemes for arbitrary grouping.

- ■ *even tags (see figure 5)*
- ■ *odd tags (see figure 6)*
- ■ *comments (see figure 7)*
- ■ *member grouping (see figure 8)*
- *some patterns*
  - *Sketchy HTML text with BODY*
    - *HTML*
      - DOCTYPE . . .
      - . . .
      - *HEAD*
        - . . . - - - *TITLE*
        - . . .
      - *HEAD*
      - *BODY*
        - . . .
      - *BODY*
    - *HTML*
  - *Sketchy HTML text with BODY*
  - ■ *Sketchy HTML text with FRAMESET (see figure 4)*
- *some patterns*

HTML mapping

Fig. 3. *HTML* mapping principles

Sketchy HTML text with FRAMESET
HTML
DOCTYPE . . .
. . .
HEAD
. . . - - - *TITLE*
. . .
HEAD
FRAMESET
COLS=" %, %"
FRAME
SRC=" " NAME=" "
FRAMESET
ROWS=" %, %"
FRAME
SRC=" " NAME=" "
FRAME
SRC=" " NAME=" "
FRAME
. . .
FRAMESET
FRAMESET
NOFRAMES
. . .
NOFRAMES
HTML
Sketchy HTML text with FRAMESET

Fig. 4. Using *FRAMESET*

even tags

Simple sketches are used to represent even tags:

EVENTAG
sketch body
EVENTAG

the sketch above is textualized to

`<EVENTAG>`
`textualized sketch body`
`</EVENTAG>`

Simple sketch heads are used to represent attributes:

EVENTAG
attributes
- - - *one or more primitive heads*
sketch body
. . .
EVENTAG

the sketch above is textualized to

`<EVENTAG attributes>`
`textualized sketch body`
`</EVENTAG>`

Primitives with comments are used for a simpler representation of even tags without attributes:

primtive text - - - *EVENTAG*

the primitive above is textualized to

`<EVENTAG>primtive text</EVENTAG>`

even tags

Fig. 5. Representation of even tags

odd tags

Simple branches are used to represent odd tags (or even tags with omitted end-tag):

TAG

branch body

the branch above is textualized to

`<TAG>`

`textualized branch body`

Simple branch heads are used to represent attributes:

TAG

attributes

- - - *one or more primitive heads*

branch body

the branch above is textualized to

`<TAG attributes>`

`textualized branch body`

odd tags

Fig. 6. Using odd tags

comments

oneline comments

Primitive (text) comments with prefix '!' represent HTML-comments:

primitive text - - - *!zzzzz*

the primitive above is textualized to

`primitive text <!zzzzz>`

Primitive head comment without prefix '!' is not subject to textualization.

oneline comments

bock comments

Loop schemes are used to represent block comments:

[TAG]

. . .

[TAG]

the sketch above (similarly a branch) is textualized as corresponding simple sketch (simple branch) enclosed between "<!" and '>' i.e. as:

<!

[TAG]

. . .

[TAG]

>

bock comments

comments

Fig. 7. Representation of commnets

member grouping

An arbitrary group of branch members may be formed by enclosing them into a module sketch or a simple tagless sketch:

[module sketch (or module branch) comment]

[sketch (or branch) head]

sketch (or branch) body

. . .

[module sketch (or module branch) comment]

[head of this (tagless) simple sketch (or branch)]

sketch (or branch) body

. . .

the module and simple sketch above are textualized to

```
[sketch (or branch) head]
textualized sketch (or branch) body
[head of this (tagless) simple sketch (or branch)]
textualized sketch (or branch) body
```

Module comments are not subject to textualization.

Beside additional structuring of sketchy HTML texts, such grouping schemes can be used also for inserting necessary sub-schemes where needed.

member grouping

Fig. 8. Groups in *HTML* document

3.3 Mapping for *LATEX*

Any sketchy *LATEX* text is described in the first branch of an outer sketch. The type of the outer sketch and its other elements (comment, head(s), additional branches) are insignificant.

For better structuring, bigger *LATEX* constructs, such as commands with parameters and environments, may be represented as sub-sketches (instead of plain-text bracing).

Module sketches are used to represent environments (see figure 10).

Branches in simple sketches are used to represent *LATEX* blocks. The sketch in figure 11 is textualized to the text in figure 9.

Note that the branch comment (as well as the branch head text) belongs to the corresponding block. The comment of the outer simple sketch is treated as a *LATEX* comment, if it is not prefixed by a backslash (like in figure 11); otherwise the sketch commment is considered a *LATEX* command.

There is no special sketchy representation for the commands without parameters. A command with parameters may be represented as a simple sketch commented by the command; every branch of such sketch represents a parameter (consider the trivial example in figure 12). Note that in this case the sketch comment

```
%simple sketch
{
module branch, encloses between { }
textulized body of branch 1
}
[
loop branch, encloses between [ ]
head of branch 2
textualized body of branch 2
]
simple branch, remains opened
textualized body of branch 3
```

Fig. 9. Textualized branches

begins with a backslash, and therefore is not considered a *LATEX* comment.
Loop and module sketches with head may be used to avoid some repetitions: during textualization of such schemes, the primitive head text is inserted in front of every non-simple branch comment. For branches of type "module", a separating blank is added to the text inserted, otherwise (for branches of type "loop") the inserted text immediately precedes the branch comment. The type of primitive head and the primitive head comment are insignificant. The module sketch with head is used to describe repetition inside the environment, otherwise a loop sketch with head should be used. For instance, the section structure of a *LATEX* document may be expressed as in figure 13. In some typical environments with repetitions (see figures 16 – 20) the (default) module head may be omitted. Arrows may be used to represent captions and labels (see figures 14, 15).
A sample general structure of a *LATEX* document:

\documentstyle{llncs}
document
title
Designing Computer Texts with {\em AMADEUS}
title
author
. . .
author
. . .
document

environment
module body
environment
Textualizes to
```
\begin{environment}
textulized module body
\end{environment}
```

Fig. 10. General environment module

simple sketch
module branch, encloses between { }
body of branch 1
loop branch, encloses between [ ]
⋆ head of branch 2
body of branch 2
simple branch, remains opened
body of branch 3
simple sketch

Fig. 11. Blocks as branches

usepackage
english,
german, estonian
babel
usepackage
Represents the text
`\usepackage[english, german, estonian]{babel}`.

Fig. 12. Command as a simple sketch

sections
⋆ \section - - - *repeated for branches*
*
{Abstract} - - - *not numbered*
In this . . .
{Previous work}
A lot of . . .
⋆ \subsection - - - *repeated*
. . .
. . .
{Future work}
. . .
*
{Acknowledgements}
- - - *not numbered*
. . .
sections

Fig. 13. Repetitions as loop sketches

figure
[tbp]
. . .
caption-text - - - *label-text*
figure

Fig. 14. Figure

table
[htbp]
caption-text - - - *label-text*
. . .
table

Fig. 15. Table

tabular
⋆ \\ - - - *default*
{|*rl*|}
hline
. . . & . . .

. . . & . . .
hline
. . . & . . .
hline
tabular

Fig. 16. Tabular

tabbing
⋆ \\ - - - *default*
. . .*quad* \\= . . . *quad* \\= *kill*
. . . \\> . . . \\> . . .

. . .
tabbing

Fig. 17. Tabbing

thebibliography
⋆ \\bibitem - - - *default*
{*99*}
{*Alb*}
Albert . . .
{*Ber*}
Berta . . .
thebibliography

Fig. 18. The bibliography

itemize
⋆ \\item - - - *default*
. . .
. . .
. . .
. . .
itemize

Fig. 19. Itemize / enumerate

description
\\item - - - *default*
[Object:]
. . .
[Instance:]
. . .
description

Fig. 20. Description

4 Future work

One of the urgent tasks is to develop automatic sketchifiers' and textualizers' generators using formally described mappings as input.
The original design of *AMADEUS* includes the possibility to use also images in the primitive texts, but this feature is not implemented yet.

References

1. J. Kiho, A. Oja. "Designing a Program Multiviewer" *Proc. 7th Nordic Workshop on Programming Theory*, Nov. 1-3, 1995, Göteborg, Programming Methodology Group Report 86, Göteborg University and Chalmers University of Technology, pp. 165-184, 1996.
2. J. Kiho, V. Hagu, M.Sutt. A Text Model for the Java-Based Model-View Multieditor. *Proc. NWPER'98 Nordic Workshop on Programming Environment Research*, June 14-16, 1998, Bergen, Informatics report No 152, Information science report No 66, pp. 221-225, 1998.
3. J. Kiho, "Diagramming techniques and sketchy programming" *Proc. 2nd Symposium on Programming Languages and Software Tools*, August 21-23, 1991, Pirkkala, Finland, Report A-1991-5, University of Tampere, pp. 78–82, 1991.
4. J. Bishop. Java Gently. Addison-Wesley, 1997, 391 p.

This article was processed using the LaTeX macro package with LLNCS style