UNIVERSITY OF TARTU
INSTITUTE OF COMPUTER SCIENCE

SKM
Sketchy Modeling of Computer Texts

Jüri Kiho

RESEARCH REPORT
2nd edition:
26th August 2004

Annotatsioon

SKM. Arvutitekstide skeem-modelleerimine
Jüri Kiho

Esitatakse arvutitekstide üldine skeem-mudel. Selle aluseks on puu, milles saab noolte abil näidata ka teatavaid tagasisidestusi. Kirjeldatakse skeem-modelleerimise printsiipe ja sellega seonduvaid funktsioone.
Skeem-modelleerimisel esitatakse suvaline arvutitekst vastava puustruktuurina – skeemtekstina, milles teksti lihtsamad osad esinevad puu tippude tekst-atribuutidena. Mudelis defineeritakse need atomaarsed tekstiosad tavateksti üldistusena, lubades ka hüpersümbolite (piltide ja omakorda skeemtekstide) kasutamist reasümbolite rollis. Lähteteksti keel (baaskeel) ja modelleeritud teksti (ekraani)vaade antakse skeemtippude vastavate lisa-atribuutidena.
Hüpersümboliteta skeemtekstide käsitlemiseks on loodud vastav *Java*-põhine süsteem *AmadeusJ*. Viimane võimaldab tekst-atribuutide, aga ka suuremate terviklike mudelielementide (skeempuu tippude) toimetamist. Toimetamisfunktsioonid on sõltumatud nii baaskeelest kui ka parajasti kasutatavast vaatest. Sptesiifilisemad, vaate ja baaskeelega seotud funktsioonid on koondatud vastavatesse eriklassidesse. Süsteem on avatud integreerimaks uusi vaateid ja baaskeeli. *AmadeusJ* toetab *Java*-, *HTML*- ja LaTeX-põhiste skeemtekstide töötlemist.
Antakse ammendav ülevaade süsteemist *AmadeusJ* ja esitatakse nimetatud kolme konkreetse arvutikeele modelleerimise põhimõtted.

Contents

Introduction

In this report, present achievements in the field of sketchy modeling and programming are outlined.
The main domain of sketchy modeling 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.

The concept of sketchy representation of computer-related texts was born about two decades ago [1, 2]. Originally, it was introduced as a convenient graphical, keyword-less and almost self-explaining notation for algorithms (see Figure 1). Since then, this notation has been widely used at the University of Tartu for teaching purposes [4, 7, 8, 15, 16, 22, 23, 24, 27, 28, 29, 37]. Soon the notation technique was applied also for control structures in computer programs, and the idea of sketchy programming was developed [3, 5]. A number of experimental support tools for sketchy programming, based on various programming languages, were implemented [9] - [14] and the sketchy programming concept was further refined [20, 21, 25].
In the 1980s, the pseudo-graphical display mode was used to depict sketchy programs. At that time more developed sketchy programming systems were built for *Fortran* [9, 17, 19] and *C* [26].
The *Fortran*-based system (*SKF*) [18] was used as the main programming tool for implementing a large DBMS for organic reactivity [34, 35]. All the programs were developed in a pseudo-graphical comprehensive sketchy form and translated (textualized) to *Fortran* source texts.
The *C*-based system (*SKC*) was already built as an integrated environ-

Euclid(a, b)
- - - *Given: integers* $a > 1,\ b > 1$
- - - *Returns: the greatest common divisor for* a *and* b

$x := a$
$y := b$

$x = y$ **?**
- - - $x = y = GCD(a, b)$
x - - - *return*
$x > y$ **?**
$x := x - y$
<- - - - - - - *repeat*
$y := y - x$

Figure 1: A sketchy algorithm

ment. It was directly used in teaching process as well as an aid for preparing algorithm-related study materials. *SKC* served also as a prototype for further change to fully graphical environments and multi-base sketchy programming systems.

In the mid of the 1990s, the idea of integrating several base languages was proposed and implemented in *C++* as a program multi-viewer *AMADEUS* [32]. In *AMADEUS*, first buds of the multi-view-multi-base concept were already present.

In parallel to practical developments, theoretical research work has been carried out, concerning automatic program structure reduction [30, 32], expressiveness of sketchy control structures [31, 33, 38] and eliminating *goto*'s while sketchifying source program code [36]. The concept of quasi-structured program as a model for sketchy programs was elaborated and relevant scientific notation introduced.

Although the potential usefulness of sketchy notation for depicting computer text structures, other than control structures, was noticed rather early [6], the works performed until late of the 1990s had been focused mainly on imperative programming languages.

An essential generalization was adopted in 1998, when a uniform sketchy text model was proposed [39]. The perfected model, suitable for any kind of structured computer texts, is specified in Chapter 1.
Originated from previous theoretical and experimental work in the field, a new project (*AmadeusJ*) was launched in 1998. It 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. In Chapter 2, the current version of *AmadeusJ* is characterized. It supports designing *Java* programs, LaTeX documents and *HTML* texts [40].

Acknowledgments

First of all, I greatly appreciate my students who have participated in all sketchy programming and modeling project works. Most of them are co-authors of our publications, and so mentioned in Bibliography. In particular, Kalle Kullman and Märt Muts contributed a lot at dawn of the sketchy programming systems, and stay still with the idea; the first version of *AMADEUS* was implemented by Asko Oja; Dmitri Brodski developed sketchifier's algorithms for solving *goto*'s, Alexey Podlipski – algorithms for textualizing *sketchy Pascal* and Roman Ustitch – algorithms for converting sketchy texts to rich *HTML* export format.
Special thanks go to Alla Jalas, with whom we worked together for many years, implementing in *sketchy Fortran* the organic reactivity DBMS. Her feedback and suggestions have been quite helpful in the process of refining general concepts.
Many thanks go to my colleagues from the Institute of Computer Science at the University of Tartu for their understanding attitude, valuable discussions and advice.
The work on sketchy programming and modeling was twice supported by Estonian Science Foundation (grant 414, 1993-1994 and grant 3343, 1998-1999). The support is greatly appreciated.

Chapter 1

Specification

1.1 The Sketchy Text Model

1.1.1 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.
The *row element* is either a *symbol* – <character, font, color>, or a hyper-symbol. (The notion "hyper-symbol" will be defined later on.)
Any sketchy text is obtained by arranging text units (and other attributes) into a tree-like structure, called the sketchy tree.

1.1.2 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.
A sample sketchy tree is depicted in Figure 1.1a, where filled circles and rectangles and unfilled rounded rectangles denote sketches, branches and primitives correspondingly. "Back-threading" is shown by dotted arrows. On the other hand, any sketchy tree can be considered as a hierarchical set of nested structures. According to such viewpoint, another picture for the same sample sketchy tree is drawn in Figure 1.1b. 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 (the only dotted arrow in Figure 1.1b "points" to b3). Such representation style is called the *sketchy view*.
Although, at a first glance, the direct pictorial view (like in Figure 1.1a) could seem to suit better, actually 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.
Note, that a subtree with the root of kind "sketch" can be considered as a sketchy tree (by relevant reduction of levels) only if it has no outgoing arrows. In the sample sketchy tree in Figure 1.1 neither the subtree with root s2 nor the subtree with root s3 are sketchy trees (because decreasing levels by 2 would yield illegal arrows).

Figure 1.1: A sketchy tree in pictorial (a) and sketchy (b) representation

1.1.3 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.

1.1.4 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 1.1.1), called the *scheme comment*. In Figure 1.1 we already used nine rows, "s1", "s2", "s3", "b1", "b2", "b3", "b4", "b5" and "b6", for scheme comments.
Every primitive head is assigned a text unit. Every primitive (i.e primitive node) is assigned a text unit. In Figure 1.2, a sketchy text obtained by attributing the sketchy tree from Figure 1.1 is depicted.
Remark. Throughout present report, for drawing sketchy texts, we use the sketchy print style, supported by *AmadeusJ* (see Chapter 2). 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.

1.1.5 Type attributes

Every node in a sketchy tree and every primitive head is assigned a *type* – a non-negative integer.
In Table 1.1 the sketchy print style drawing conventions for some scheme types are given. So, the sample sketchy text in Figure 1.2 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 1.2, the primitive

s1
Example - - - *head of sketch s1, one primitive head*
b1
B1B1 - - - *head of branch b1, one primitive head*
this text unit
consists of two rows - - - *p1*
arrow to s1 - - - *p2*
b2
this text unit consists of one row - - - *p3*
s2
b4
arrow to s1 - - - *p4*
b5
s2
b3
s3
⋆ n - - - *first primitive head for sketch s3*
⋆ m - - - *second primitive head for sketch s3*
b6
some text unit - - - *p5*
some text unit - - - *p6*
<- - - - - arrow to b3 - - - *p7*
arrow to s3 - - - *p8*
s3
s1

Figure 1.2: A sketchy text as an attributed sketchy tree

Table 1.1: 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)

heads of sketch s3). The full set of primitive type prefixes for the sketchy view styles, supported by *AmadeusJ*, is given in Chapter 2.

1.1.6 Other attributes

The rest of attributes

- views
- icon-tags
- base languages

are for sketches and branches.
Every scheme is assigned a *view*, which defines its display style – screen planning and drawing for this particular scheme.
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 1.3, the sample sketchy text from Figure 1.2 with *true* icon-tags for branches b1 and b2 is depicted.

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. The base language attribute is not directly reflected in any view.

s1
Example - - - *head of sketch s1, one primitive head*
▪ *b1*
▪ *b2*
b3
s3
⋆ n - - - *first primitive head for sketch s3*
⋆ m - - - *second primitive head for sketch s3*
b6
some text unit - - - *p5*
some text unit - - - *p6*
<- - - - - - arrow to b3 - - - *p7*
←— arrow to s3 - - - *p8*
s3
s1

Figure 1.3: A sketchy text with icon-tags set for branches b1 and b2

1.1.7 Generalization: hyper-symbols

A more general and flexible approach to sketchy modeling is obtained by introducing hyper-symbols as row elements. It means that beside usual "plain" symbols more complex items can be included into rows.
The hyper-symbol is either a *picture* – a pair <*icon-tag*, *image*>, or a *hyper-element* – a sketchy text.
Inclusion sketchy texts as hyper-symbols involves certain structure for sketchy texts under consideration. We say that sketchy text s *directly includes* sketchy text s' if $s = s'$ or s' occurs in (some row of) s as a hyper-symbol.
A sketchy text s *includes* sketchy text s' if there exists a sequence

$$s_1 = s, s_2, \ldots, s_{n-1}, s_n = s'$$

such that s_i directly includes s_{i+1} $(i = 1, \ldots, n-1)$.
Any recursive inclusion is forbidden: a sketchy text s can be used as a hyper-symbol in (some row of) another sketchy text s' only if s does not include s'.

1.1.8 The modeling language

Below, the BNF syntax of sketchy texts is given. It does not reflect the constraints imposed on arrows' endpoint levels and the usage of hyper-elements. Some helpful additional non-terminals (*sketch body*, *branch body*, *member*) are introduced.

Meta-symbols: { } – "0 or more".
Terminal symbols: *non-negative_integer*, *character*, *font*, *color*, *image*, *true*, *false*, *view*, *base_language*.

sketchy_text ::= sketch
scheme ::= sketch | branch
sketch ::= sketch_body scheme_attributes
sketch_body ::= {branch}
branch ::= branch_body scheme_attributes
branch_body ::= {member}
member ::= sketch | primitive_member
primitive_member ::= primitive | arrow
primitive ::= type text_unit
arrow ::= primitive endpoint_level
endpoint_level ::= *non-negative_integer*
type ::= *non-negative_integer*
text_unit ::= {row} comment
comment ::= row
row ::= {row_element}
row_element ::= symbol | hyper-symbol
symbol ::= *character font color*
hyper-symbol ::= picture | hyper-element
picture ::= icon-tag *image*
hyper-element ::= sketchy_text
scheme_attributes ::= type comment head icon-tag *view base_language*
head ::= {primitive_head}
primitive_head ::= primitive
icon-tag ::= *true* | *false*

1.2 Views

View is one of the scheme attributes. As pointed out earlier, 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.).

Any view defines two activities: planning and drawing. Planning scheme s involves planning all its constituents (using, in particular, the views for sub-sketches and sub-branches) and calculating necessary screen area for s. Drawing scheme s displays s in the planned screen area. If s is a sketch, beside the complementary graphical elements, its comment and head, and also all its sub-branches are drawn (using, in turn, their views). If s is a branch, beside the complementary graphical elements, its comment and head, and also all its members are drawn (using, in turn, views of the member-sketches). 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. Displaying a hyper-symbol in a row means positioning either an icon or a picture or a sketchy text (drawn by its view) in the row.

1.3 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 and branches of certain type, their heads and bodies and body members. Particular examples of computer text modeling are given in Chapter 2.
As a simple example for a data domain, let us consider a table-data description language. Every portion of data is a two-dimensional table, containing one head-row of text items, and some data-rows of numeric items; a table caption may also be given.
There are many possibilities to model such kind of data. Two solutions are described in the next page. By default, all types are 0 in both cases. For this specific data type, the sketchy view (see Figure 1.4 and Figure 1.5) might not be practically suitable for table-data handling. An appropriate specific view, say, *table view*, could be implemented and assigned to such kind of sketchy texts. The table view might be designed to display the data in the normal table form, such as Table 1.2. Independent of the current view, any such table may be edited using common tools. For instance, in the case of the first model, inserting a branch means inserting a column.
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. For instance, one can easily design a sketchy notation for representing hierarchical structures for some kind of organization, like in Figure 1.6. Such kind of mapping is rather obvious and self-explaining. If necessary, arrows may be used to denote reporting-paths from lower units to higher ones. For more convenient handling such structures, an appropriate tree-diagram view could also be supplied.

Table 1.2: A sample

x	f(x)	g(x)
2.0	0.59	0.05
0.1	6.98	0.19
-1.0	17.02	0.23

Sketchy model I for table-data: a table with m data-rows and n columns is sketch s, containing n branches $b_1, b_2, \ldots, b_n$; the table caption is the comment of sketch s; i-th column is branch b_i, containing m primitive members; the text unit for j-th member is j-th numeric item in the column converted to string; comment of b_i is i-th text item from the head-row.

Sketchy model II for table-data: a table with m data-rows and n columns is sketch s, containing $m+1$ branches $b_0, b_1, \ldots, b_m$; the table caption is comment of sketch s; the head-row is branch b_0, containing n primitive members; the text unit for j-th member is j-th text item in the head-row; i-th data-row is branch b_i, containing n primitive members; the text unit for j-th member is j-th numeric item in the data-row converted to string.

A sample
x
2.0
0.1
-1.0
f(x)
0.59
6.98
17.02
g(x)
0.05
0.19
0.23
A sample

Figure 1.4: Sketchy table I

A sample

x
f(x)
g(x)

2.0
0.59
0.05

0.1
6.98
0.19

-1.0
17.02
0.23
A sample

Figure 1.5: Sketchy table II

Below, some practical design recommendations for defining sketchy mappings are given.
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 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.
The rest of recommendations are related to designing sketchy mappings for control structures in procedural programming languages. The following principles should be kept in mind.
The control flow in an executable branch is regulated by arrows and conditions. A *condition* is a primitive of special type (usually of type 1) which text unit is a boolean expression in the base language. It has the role of a semaphore – if the boolean expression is evaluated to *false*, a jump occurs: executing proceeds just form behind the nearest downward arrow in the branch; if such arrow is absent, execution of the branch terminates.
An arrow terminates execution of the scheme (i. e. the sketch or the branch), "pointed" to by the arrow. Any loop is represented by a sketch of special type (usually of type 1) which has a single branch (representing the loop body). After (normal) terminating the branch, execution of the sketch resumes, i.e. the branch is executed repeatedly. For the head of the loop (if present), a primitive head of special type (usually of type 1) is used, which text unit is the loop head in the base language (and controls repetitions of the loop branch). Selection constructs, such as *switch* and *try* statements are represented by multi-branch sketches (usually of type 0), each branch representing a case (for *switch* statements) or try/catch blocks (for *try* statements). The heading data may be represented by scheme heads. For instance, for a *switch* statement, the sketch head could contain the switch-expression, the branch heads – the case labels; for a *try* statement, the catch-branch heads could contain the formal parameters.
Also, the branches are just the proper constructs for modeling parts of algorithms and programs to be executed in parallel.

- *A typical organization of a IS system factory*
  - *GENERAL MANAGER (system factory)*
    - *DEPARTMENT MANAGER (administration)*
    - *DEPARTMENT MANAGER (finance)*
    - *DEPARTMENT MANAGER (purchasing)*
    - *DEPARTMENT MANAGER (engineering management)*
    - *DEPARTMENT MANAGER (production management)*
    - *DEPARTMENT MANAGER (quality assurance)*
    - *DEPARTMENT MANAGER (application system)*
      - *SECTION MANAGER (system)*
        - UNIT MANAGER - - - *system analysis*
        - UNIT MANAGER - - - *project integration*
      - *SECTION MANAGER (system)*
    - *DEPARTMENT MANAGER (application system)*
    - *DEPARTMENT MANAGER (basic software)*
      - *SECTION MANAGER (production management)*
      - *SECTION MANAGER (quality management)*
      - *SECTION MANAGER (OS)*
        - UNIT MANAGER - - - *R&D*
        - UNIT MANAGER - - - *design*
        - UNIT MANAGER - - - *programming*
        - UNIT MANAGER - - - *testing*
        - UNIT MANAGER - - - *maintenance*
      - *SECTION MANAGER (OS)*
      - *SECTION MANAGER (Language)*
        - UNIT MANAGER - - - *R&D*
        - UNIT MANAGER - - - *design*
        - UNIT MANAGER - - - *programming*
        - UNIT MANAGER - - - *testing*
        - UNIT MANAGER - - - *maintenance*
      - *SECTION MANAGER (Language)*
    - *DEPARTMENT MANAGER (basic software)*
  - *GENERAL MANAGER (system factory)*
- *A typical organization of a IS system factory*

Figure 1.6: An organization structure

1.4 The plain sketchy text

In this section a trivial mapping for plain computer texts is introduced.
A text unit is called the *plain text unit* if it consists of a single row without hyper-symbols and its comment is an empty row.
A primitive of type 0 is called the *plain primitive* if its text attribute is a plain text unit.
A headless branch of type 0 is called the *plain branch* if all of its members are plain primitives.
A headless sketch of type 0 with a single plain branch is called the *plain sketch*.
A plain sketch is called the *plain sketchy text*.
The trivial mapping for plain computer texts is obvious: the (trivial) sketchy representation for plain source t consisting of n rows is a plain sketchy text with a single branch b consisting of n primitives, and i-th primitive is assigned a plain text unit containing i-th row of t ($i = 1, \dots, n$). Arbitrary colors and fonts may be used while converting plain characters to row symbols.
In this sense, any plain sketchy text may be considered as the trivial representation of a plain text.

1.5 Sketchifying

Given a sketchy mapping from non-empty base language L to the sketchy text model, the method for translating any source in L into the corresponding sketchy text, is called the *sketchifier*. The translating process is called *sketchifying*.
In the resulting sketchy text, all schemes are assigned L as the base language attribute.

1.6 Textualizing

Generally speaking, textualizing is reverse to sketchifying. i. e. means translating some given sketchy text into semantically equivalent source text. Nevertheless, certain specific aspects should be considered.
First, textualizing is meaningful only if the base language describes plain computer texts. Otherwise, i. e. if the base language describes some "inner" data structures, the process is actually *compiling* rather than textualizing.

In the case of empty base language, neither textualizing nor compiling cannot be performed.
Second, sub-schemes may have different base language attributes, so representing different mappings of computer texts. Therefore, textualizing should be carried out scheme by scheme, using, possibly, different textualizers for different schemes.
We define *textualizing* as a process, aiming to translate a text-based scheme into a (semantically equivalent) plain sketchy text. Corresponding methods are called the *textualizers*.

1.7 Transforming

Any method for changing sketchy texts is called *transforming*.
Some typical transformations, like reducing and normalizing, will be discussed in Chapter 2. Actually, both sketchifying and textualizing may be implemented as transformations too – "from plain sketchy text" or "into plain sketchy text" correspondingly.

1.8 Solving

It is often desirable to have integrated tools for direct solving sketchy models. For the models based on common programming languages, it means that corresponding tools for immediate compiling, debugging and running are available.
Similar model solving facilities may be designed and implemented for other kinds of sketchy texts too. Such tools perform calculations using given sketchy texts as input data.

Chapter 2

Implementation

As a support tool for the technique of sketchy modeling – a working prototype of a multi-view multi-base sketchy editor, named *AmadeusJ*, has been developed.

2.1 *AmadeusJ*: an overview

AmadeusJ is a *Java*-based multi-view multi-language general-purpose system for handling computer texts, such as program sources, LaTeX documents, *HTML* texts et al. It supports graphical representations of 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 sketchy text. The sketchy text model offers rather general opportunities for reasonable text structuring. In the following sections, main features of the multi-editor are described, and 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.
The main restriction, compared to the general specification, relates to text units. In the current implementation, the use of hyper-symbols is excluded. Although the notion *image* was included into the sketchy text model for *AmadeusJ*, it is not supported by its current implementation. No model solving facilities are integrated into *AmadeusJ*.
AmadeusJ is currently under development. The last version as a set of *Java*-classes is available at `http://www.cs.ut.ee/~kiho`.

2.2 Main features of ***AmadeusJ***

2.2.1 The modeling language

Meta-symbols: { } – "0 or more"
Terminal 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 in use. Sketches of these types are called the *simple sketch*, the *loop sketch*, and the *module sketch* correspondingly.

2.2.2 Views

For screen display, *AmadeusJ* fully supports the sketchy view (named *Sketches*), and its modification (*Sketches+*). The last one is basically the same as the first one, but sufficiently small sketches are arranged into one screen row in the case of view *Sketchy+*.

Table 2.1: Reflection of element types in the sketchy view

Element	Type	Displayed as
Sketch	0	left bracket
Sketch	1	double left bracket
Sketch	2	boldface left bracket
Branch	0	separator — (single)
Branch	1	separator = (double)
Branch	2	separator **—** (single, bold)
Primitive head	0	prefix " "
Primitive head	1	prefix "*"
Primitive head	2	prefix "??"
Primitive head	3	prefix "?"
Primitive head	4	prefix "?"
Primitive head	5	prefix "!!"
Primitive head	6	prefix "!"
Primitive head	7	prefix "!"
Primitive	0	prefix " "
Primitive	1	prefix "?"

In Table 2.1, the tags for the sketchy view, reflecting the types of sketchy text elements, are listed.

A simple tree-view (named *STree*) is designed for handling simpler tree structures. In this case, every node (say, node x) should be represented as a sketch, containing a single branch (b). The successors of node x are, in turn, the member-sketches of the branch b. The comment of a sketch is considered a node comment. The node label is represented as the head in corresponding sketch.

The other views are experimental (and not fully supported). In particular, the view *Boxes* may be used to see sketches and branches of the current sketchy text as a collection of nested rectangles.

Remark. In the sketchy view as well as in the sketchy print style, the scheme icon may be shown together with a text row from the scheme. If the scheme comment is not empty, then the comment is shown. Otherwise, if the scheme head is not empty, then the first row of the first primitive head is shown.

2.2.3 Other features

Multi-editor *AmadeusJ* 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". The multi-editor 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 attribute 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. The features are more closely discussed in Section 2.7.
Remark. Currently, only the insert edit mode is supported. Some miscellaneous key-functions ("Page Up", "Page Down", "Home", "End") are not implemented yet.

2.3 Designing computer texts

AmadeusJ currently supports three base languages: *Java*, *HTML*, LaTeX. To add a new one, a 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, *AmadeusJ* is ready to be used for designing and developing computer texts in this base language too.
In the following, three design solutions for mapping computer texts onto the sketchy text model are considered. Beside obtaining well-structured texts, the redundant typing of various end-tags, keywords and group separators is excluded. In the current version of *AmadeusJ*, the transforming methods assume that all schemes in the sketchy text (selected for transforming) have the same base language attribute as the main sketch of the selection.

2.4 Modeling *Java* sources

Any sketchy *Java*-program is described in the first branch of an outer sketch. The type of the outer sketch and its other elements (comment, head, additional branches) are insignificant. For better structuring, bigger program constructs, such as classes, modules, methods, functions, flow controls etc. may be represented as sub-sketches (instead of plain-text bracing or keywording).

2.4.1 Groupings

A headless module sketch denotes an arbitrary *group*. It should contain a single branch and there should not be any arrow "pointing" to that sketch. It is opened automatically prior to textualization: it will be replaced by the sequence of its branch members.
A module sketch with a primitive head of type 1 denotes the *prefixed group*. It should contain a single branch and there should not be any arrow "pointing" to that sketch. It is opened automatically prior to textualization: it will be replaced by the sequence of its branch members, and the text unit of every primitive member of type 0 will be prefixed by the text unit from the primitive head.
In addition, any sketch can be marked as a *comment group*, i. e. "commented out", by prefixing its comment by two slashes ("//"). Such sketch will be skipped during textualization.
In Figure 2.2 the result of textualizing the source from Figure 2.1 is shown.

2.4.2 Classes

Any *Java* class declaration is mapped as a module sketch with head (see Figure 2.3). Its body contains one headless branch (of type 0), representing the class body declaration (without enclosing braces "{","}"). The head contains the head-text preceding the class body, i. e. modifiers, keyword *class*, the superclass name and the interface list. All primitive heads are of type 0. The head-text may be arbitrarily cut and divided between primitive heads' text units.

```
. . .
states
⋆ static final int STATE_

START = 0;
PROCESSING = 1;
END = 9;
states
. . .
initialize
int state = STATE_START;
int x = -1;
double a = 0.0;
initialize
// initialize
int state = STATE_START;
int x = 0;
double a = -1.0;
// initialize
. . .
```

Figure 2.1: Groups in sketchy *Java*

```
This text is generated by AMADEUS
. . .
static final int STATE_START = 0;
static final int STATE_PROCESSING = 1;
static final int STATE_END = 9;
. . .
int state = STATE_START;
int x = -1;
double a = 0.0;
. . .
This text is generated by AMADEUS
```

Figure 2.2: Groups in *Java*-source after textualization

B
public
class B extends A implements I1, I2, I3

class body
B

Figure 2.3: A sketchified class declaration

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

method body
BinarySearch

Figure 2.4: A sketchified method declaration

2.4.3 Methods

Any *Java* method declaration is mapped as a module sketch with head (see Figure 2.4). Its body contains one headless branch (of type 0), representing the method body declaration (without enclosing braces "{","}"). The head contains the head-text preceding the method body, i. e. modifiers, the return type, the method name, the parenthesized formal parameters' list and the throws clause. All primitive heads are of type 0. The head-text may be arbitrarily cut and divided between primitive heads' text units.

2.4.4 ***Try*** statement

Any *try* statement is mapped as a simple sketch with head (see Figure 2.5). Its head contains a single primitive head of type 5 with an empty text unit. The body of the try-sketch contains several branches. The first branch is headless and represents the try-block. If the finally-block is present, it is represented by the last branch, which has a single primitive head of type 7 with an empty text unit. Every other branch has a primitive head of type 6 and represents the catch-clause: its body represents the catch-block and the

primitive head text unit contains the formal parameter.
Try-, catch- and finally-blocks are given without enclosing braces "{","}".

2.4.5 *Switch* statement

Any *switch* statement is mapped as a simple sketch with head (see Figure 2.6). Its head contains a single primitive head of type 2 with a text unit containing the switch-expression. The body of the switch-sketch contains several branches, representing the case-clauses. Each such branch has a primitive head of type 3, containing the case label (together with ":") as text unit; the branch body represents the case-body. If the default-clause is present, it is represented by the last branch, which has a single primitive head of type 4 with empty label (":").

2.4.6 Loop statements

Any *for* statement is mapped as a loop sketch with head (see Figure 2.7). Its head contains a single primitive head of type 1 with a text unit containing the for-expression (without enclosing parentheses). Its single branch is of type 0 and represents the loop body (without enclosing braces "{","}").
Any *while* statement is mapped as a *for* statement with for-expression "; while-expression;" (see Figure 2.8).

trying
!!

■ *actions to try*

! (Exception1 e)
■ *actions when Exception1*

! (Exception2 e)
■ *actions when Exception2*

!
■ *the final actions*
trying

Figure 2.5: A sketchified *try* statement

??f(x)

? 1:
■ *when f(x) == 1*

? 4:
■ *when f(x) == 4*

? :
■ *otherwise*

Figure 2.6: A sketchified *switch* statement

⋆ int i=0; i<n; i++

■ *loop body*

Figure 2.7: A sketchified *for* statement

⋆ ; notFound() ;

■ *loop body*

Figure 2.8: A sketchified *while* statement

■ *loop body*
Math.abs(y) >= eps **?**

Figure 2.9: A sketchified *do* statement

synchronized(e)

■ *body*

Figure 2.10: A sketchified *synchronized* statement

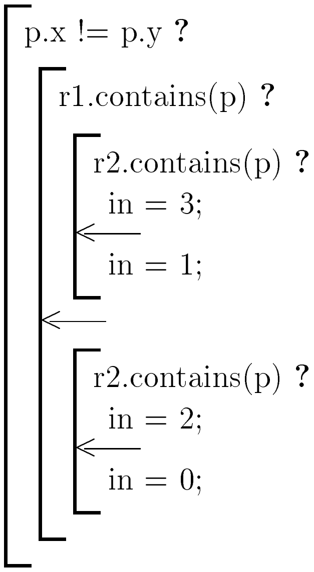

Figure 2.11: Nested *if* statements

Any *do* statement is mapped as a headless loop sketch which single branch contains the body-statements followed by a primitive member of type 1. The text unit of this last primitive-member represents the while-expression (see Figure 2.9).

2.4.7 *Synchronized* statement

Any *synchronized* statement is mapped as a simple sketch with head (see Figure 2.10). Its head contains a single primitive head of type 0 with a text unit containing "synchronized(expression)". Its single branch is of type 0 and represents the statement body (without enclosing braces "{","}").

2.4.8 *If* statement

Any *if* statement is represented by a headless simple sketch with a single branch of type 0. The first member of the branch is a primitive of type 1 with text unit representing the if-expression (without enclosing parentheses). The second member represents the then-statement. The else-part (if present) is represented by two following members: the third one is an arrow "pointing" to the simple sketch and the fourth one represents the else-statement. In Figure 2.11 the sketchy representation of the *Java* statement.

```
if(p.x != p.y){
        if(r1.contains(p)){
              if(r2.contains(p)) in = 3; else in = 1;
        }
        else if(r2.contains(p)) in = 2; else in = 0;
}
```

is depicted.

2.4.9 Compound sequence

Any *Java* construct, other than class body, method body, try-block, catch-block, finally-block, loop body and body of synchronized statement, enclosed between braces "{" and "}" is called the *compound sequence.*
Java blocks in role of statements, and array initializers are just compound sequences.
Any compound sequence is represented by a headless simple sketch with single branch of type 0. The branch contains the elements of the compound sequence.

2.4.10 *Return*, *break* and *continue* statements

Any *return* statement is represented by a headless simple sketch with a single branch of type 0. The branch contains just one arrow which "points" to corresponding method-sketch and which text unit contains the return-expression.
Any *break* statement is represented by a headless simple sketch with single branch of type 0. The branch contains just one arrow which "points" to the sketch to be terminated.
Any *continue* statement is represented by a headless simple sketch with single branch of type 0. The branch contains just one arrow which "points" to the branch to be terminated. This branch is just the single branch of some (outer) loop sketch.
A sample method containing arrows for *return*, *break* and *continue* is given in Figure 2.12.

```
static long prs(int[ ][ ] a)
 - - - Given: table a
 - - - Returns: product of table row sums
 - - - Constraints: 1) rows containing a zero element are skipped;
 - - - 2) if an empty row occurs, it is skipped together with the rest of rows

long res = 1;

⋆ int i = 0; i < a.length; i++

    a[i].length == 0 ?
        - - - break

    long sum = 0;

    ⋆ int j = 0; j < a[i].length; j++

        a[i][j] == 0 ?
            - - - continue outer loop

        sum += a[i][j];

    sum == 0 ?
        0   - - - returns 0

    res *= sum;

res   - - - returns res
```

Figure 2.12: Arrows for *return*, *break* and *continue*

2.4.11 Reducing

It is assumed that the semantics of *Java*-based sketchy texts is consistent with the general principles for mapping control structures, suggested in Section 1.3. Therefore several semantics-preserving transformations over *Java*-based sketchy texts can be considered. One of such procedures is *reducing*, which aims to simplify sketchy control structures. In *AmadeusJ* an automatic reducer is implemented. It performs several simplifications; in particular, it attempts to open as many control sketches as possible. Only the sketches having empty comment or a comment prefixed by two dots ".." are subject to opening.

In Figure 2.13, the result of reducing the sketchy text from Figure 2.12 is shown.

While developing a sketchy *Java* program the developer needs not to take care of producing the shortest control structures. He or she can use whatever (semantically proper) structures he or she likes. At any point of the development process the reducer may be applied to obtain necessary optimization. The reducing algorithm is based on theoretical results about reducibility [36].

2.4.12 Sketchifying

In *AmadeusJ*, it is possible to transform a given *Java* source into the sketchy form. First, the source text should be read from a file, activating menu items "File + Read text" – the corresponding plain sketchy text will be built. After that, the latter should be selected and base language attribute "Java" assigned. Activating menu items "Tools + Sketchify" applies the sketchifier, and the result sketchy text will be built.

The sketchifier uses directly the above rules for mapping class declarations, method declarations, control statements etc.

End-comments "// . . ." are represented as comments in text units.

Any block comment "/* . . .*/" is at first placed in separate rows and then replaced by a group of primitives. Every primitive of the group has an empty text unit with comment, containing corresponding row of the block comment.

```
static long prs(int[ ][ ] a)
 - - - Given: table a
 - - - Returns: product of table row sums
 - - - Constraints: 1) rows containing a zero element are skipped;
 - - - 2) if an empty row occurs, it is skipped together with the rest of rows
long res = 1;
 ⋆ int i = 0; i < a.length; i++
   a[i].length == 0 ?
 <——   - - - break
   long sum = 0;
    ⋆ int j = 0; j < a[i].length; j++
      a[i][j] == 0 ?
 <------   - - - continue outer loop
      sum += a[i][j];
   sum == 0 ?
<——— 0   - - - returns 0
   res *= sum;
<—— res   - - - returns res
```

Figure 2.13: A reduced sketchy text

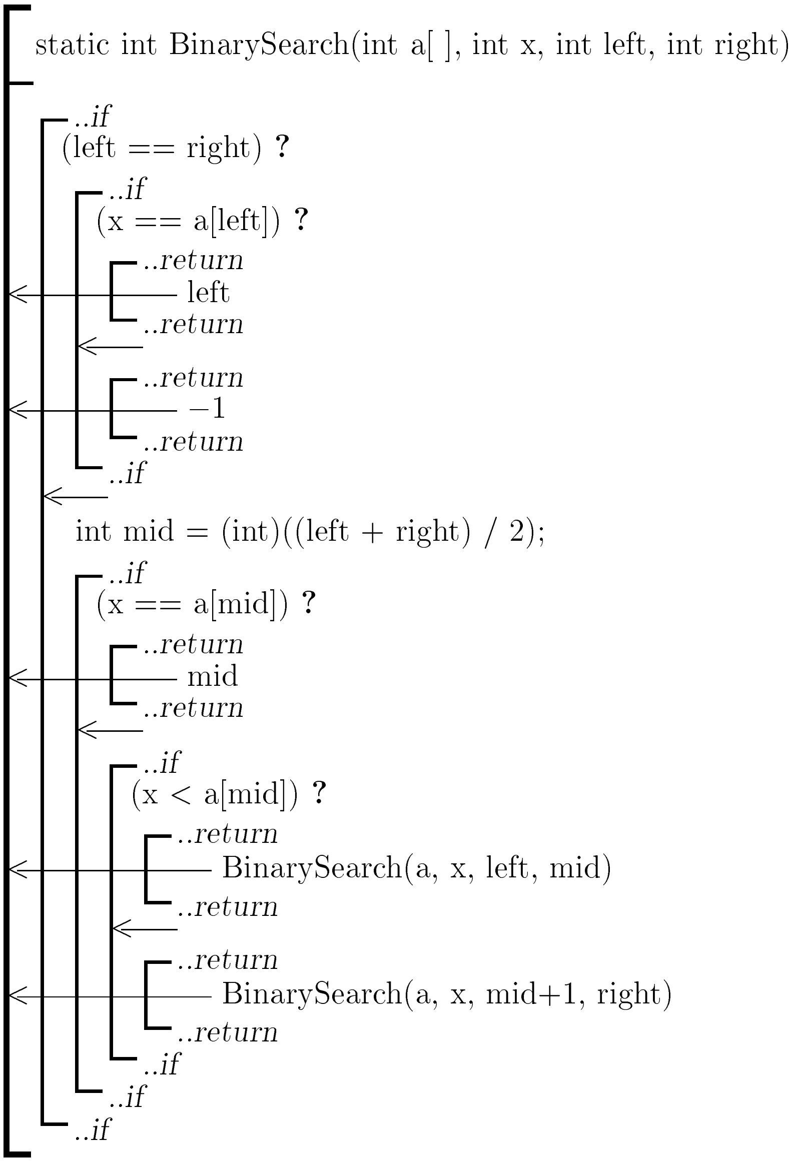

Figure 2.14: A sample sketchified method declaration

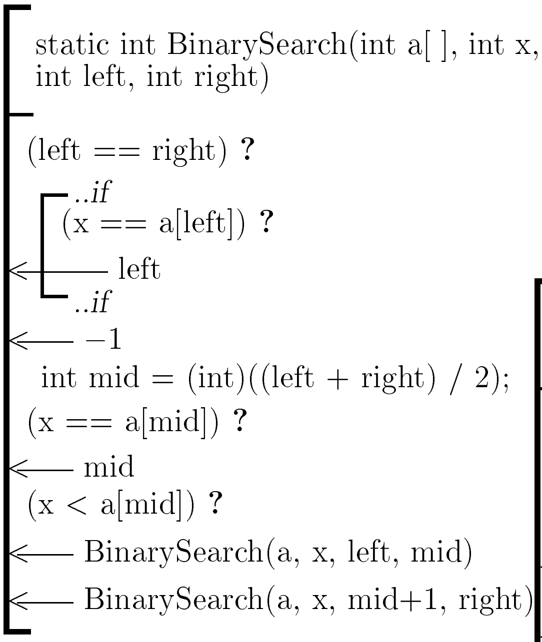

Figure 2.15: The reduced method declaration

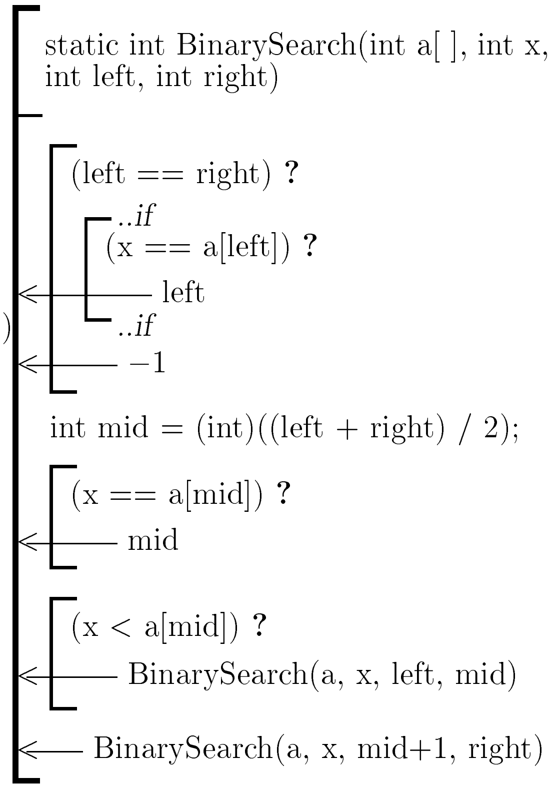

Figure 2.16: The normalized method declaration

Every *Java-Doc* comment "/** . . .*/ is represented as a (headless) group, which branch contains rows of the comment as primitives of type 0: each primitive has an empty text unit with corresponding comment. The group-sketch has comment "Javadoc" and its icon-tag is set to *true*. Often, some manual adjustments of comments, as well as reducing the result from the sketchifier, are needed.
The result of sketchifying a method from a *Java* source is in Figure 2.14. As one can see, the sketchifier comments all created sketches. The output from the reducer for this sketchy method is given in Figure 2.15.
Constraints. Due to some deficiencies of the current version of *Java*-sketchifier, the following constraints are imposed on sources. In *if* statement, the then-part must not be another *if* statement in turn (yet "if(c1){if(c2) ...}..." is allowed). Only spaces are allowed between keywords *else if*. Anonymous classes (as actual parameters) remain unsketchified. The source *Java* program is supposed to be syntactically correct.

2.4.13 Normalizing

Opposite to reducing, normalizing is a transformation which introduces additional control sketches. It is usually performed prior to textualizing (see the next section).
Given a *Java*-based sketchy text, the normalizer transforms it to the form, where every arrow occurs just as the last primitive of a branch. Let B be a branch and a the first member-arrow in branch B. If a is not the last member, a normalizing step is applicable: there must exist a condition (a primitive of type 1) preceding arrow a (otherwise the members after a would be unreachable). Let c be the first condition in B. Then the normalizer encloses the sequence of members $c \ldots a$ into an extra control sketch: the sequence is replaced be a simple sketch which single branch contains the members $c \ldots a$. The "destination" scheme for arrow a remains the same. Such normalizing step is performed for any branch (in all control sketches) while applicable.
Normalizing the sketchy text from Figure 2.15 yields the sketchy text in Figure 2.16.
The normalizer inserts special label-comments for those control sketches which are "pointed" to by some (break or continue) arrows. Normalizing performs under strong equivalence, the result is reducible to origin.

2.4.14 Textualizing

Given a *Java*-based sketchy text the corresponding textualizer translates it into pure *Java* code.
First, all comments are deleted. Second, all groups are opened. Third, normalization is performed. Fourth, all sketches are replaced by their textual equivalents; at that, arrows are replaced by *return*, *break* or *continue* statements. Due to preliminary normalization, *else* is never needed.
The result is a plain sketchy text, which may be saved as plain text by activating the menu items "File + Write text".

2.5 Modeling ***HTML*** texts

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, additional branches) are insignificant. For better structuring, tagged elements of the source *HTML* text may be represented as simple sub-schemes (instead of plain-text tagging). Loop schemes can be used for block comments, and module schemes for arbitrary groupings. All primitives and primitive heads are of type 0. Arrows are not used.
AmadeusJ supports designing (writing) new *HTML* texts rather than handling existing ones. The user is offered convenient tools for developing *HTML* texts in a comprehensive sketchy mode. The prepared sketchy text can be automatically textualized into plain *HTML* text. Though, less powerful sketchifier is usable, which performs some partial sketchifying for given plain *HTML* texts. Below, the sketchy *HTML* constructs are defined and corresponding textualizing procedures are described.
In Figure 2.17 and Figure 2.18 two general patterns for sketchy *HTML* texts are depicted.

Sketchy HTML text with body
HTML
DOCTYPE . . .
. . .
HEAD
. . . - - - *TITLE*
HEAD
BODY
. . .
BODY
HTML
Sketchy HTML text with body

Figure 2.17: A pattern for sketchy *HTML* text

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

Figure 2.18: A pattern for framed sketchy *HTML* text

2.5.1 Even tags

A simpler even tags' group without attributes may be represented by a primitive, which text unit contains the group body and comment contains the start-tag. For instance, the primitive
Subsection 5 - - - $H2$
will be textualized to
<H2>Subsection 5 </H2>
Commented simple sketches are used to represent bigger even tags' groups. Any such group is a simple sketch, commented by the start-tag. For instance, the sketch

HEAD
Welcome ! - - - *TITLE*
HEAD

will be textualized to
<HEAD> <TITLE> Welcome ! </TITLE> </HEAD>
The simple sketch head is used to represent attributes of the group. For instance, the sketch

A
HREF=MAILTO:kiho@cs.ut.ee

e-mail
A

will be textualized to
<A HREF=MAILTO:kiho@cs.ut.ee> e-mail </A>
The attributes' text parts may be arbitrarily allocated between the primitive heads' text units. The comments of the text units are ignored by the textualizer.

2.5.2 Odd tags

Commented branches are used to represent odd tags' (or even tags' with omitted end-tags) groups. While textualizing a commented branch, the textualizer encloses its comment together with its head text (if the head is present) into < >. So, the branch comment should contain the tag and the branch head – the attributes of the group. For instance, the sketch in Figure 2.19 (containing only headless branches) will be textualized to

DL
DT
Object 1
DD
the main object
DT
Object 2
DD
the auxiliary object
DL

Figure 2.19: Commented branches

<DL>
<DT> Object 1 <DD> the main object
<DT> Object 2 <DD> an auxiliary object
</DL>

2.5.3 Groupings

Uncommented schemes are used to make arbitrary (uncommented) groups in sketchy*HTML* texts.
The result of textualizing of an uncommented scheme consists of the head-text (if the head is not empty) without primitive heads' comments and the result of textualizing the body of the scheme. If the scheme is of type 1, the text obtained will be enclosed between "<!" and ">" (see the next section).
To make a commented group, a module sketch (of type 2) must be used. Any module sketch is textualized in the same manner as a simple sketch, but the comment is ignored.

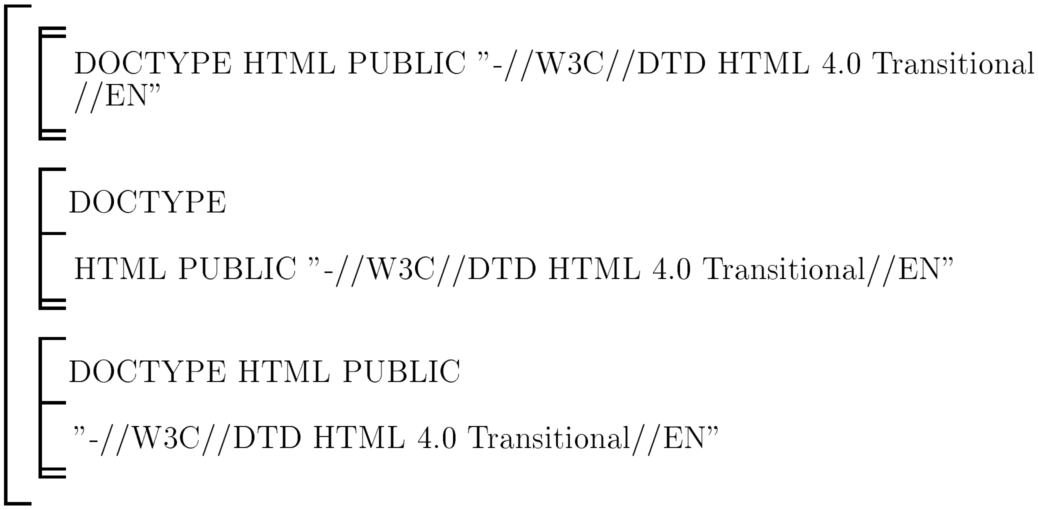

Figure 2.20: Possible expressions for one document type

2.5.4 ***HTML* comments**

Any primitive text unit comment, prefixed by ”!” is considered as a *HTML* one-row comment (instead of being even tag). For instance, the primitive
Commented text - - - !comment
will be textualized to
Commented text<!comment>
For block comments, schemes of type 1 are used. Any sketch or branch of type 1 is textualized just in the same way as scheme of type 0, but the result text is enclosed between ”<!” and ”>”.
In Figure 2.20 several possibilities to express in the sketchy form the same document type definition,
<!DOCTYPE HTML PUBLIC ”-//W3C//DTD HTML 4.0 Transitional //EN”>
are given.

2.5.5 Sketchifying

Given a *HTML* text (as plain sketchy text) the sketchifier builds corresponding sketchy text where just even tags groups are represented by commented simple sketches.

2.6 Modeling LaTeX texts

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, 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). *AmadeusJ* supports designing (writing) new LaTeX texts rather than handling existing ones. The user is offered convenient tools for developing LaTeX texts in a comprehensive sketchy mode. The prepared sketchy text can be automatically textualized into plain LaTeX text. No proper LaTeX sketchifier is currently available.

2.6.1 Environments

Module sketches are used to represent environments. An environment named "xxx" is a module sketch with comment "xxx". For instance, the module

center
module body
center

will be textualized to

```
\begin{center}
textualized module body
\end{center}
```

Textualizing the sketch body means textualizing all of its branches.

2.6.2 Groups

In LaTeX, two kinds of groups are commonly used. First, a part of text enclosed between braces "{" and "}" – the *brace-group*; second, a part of text enclosed between brackets "[" and "]" – the *bracket-group*. In addition, we shall consider the *open-group* which is an arbitrary text part, the user likes to be distinguished.

For sketchy representing (bigger) groups, the branches of simple sketches are used: the branch of type 0 – for open group, the branch of type 1 – for bracket-group, and the branch of type 2 – for brace-group. Once represented as a branch, brackets and braces are not needed – after textualizing the branch body, the result will be properly enclosed. More precisely, the result of textualizing of a branch of a simple sketch consists of the branch comment,

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

Figure 2.21: Groups as branches

head-text units (if head is present) and the result of textualizing the body of the branch. At that, if the branch is of type 1, the text obtained will be enclosed between "[" and "]", if of type 2 – between "{" and "}". Comments of text units are prefixed by "%" and also included in the result.
A collection of groups (a group of groups) may be represented as a simple sketch with several branches. The comment of such sketch must not start by "\" (otherwise it is considered as a command with parameters, see the next section). While textualizing, the non-empty comment is prefixed by "%" and inserted in front of the textualized branches. In Figure 2.21 a sample collection of branches is shown. The result of its textualization is given in Figure 2.22.

2.6.3 Commands

There is no special sketchy notation for commands without parameters.
Any command with parameters can be represented as a simple sketch which branches represent the parameters. The command name (together with the prefix "\") is given as the sketch comment. For instance,

```
%simple sketch
{
module branch, encloses between { }
textualized 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
```

Figure 2.22: Textualized branches

usepackage
english,
german, estonian
babel
usepackage

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

2.6.4 Repetitions

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 2, a separating blank is added to the text inserted, otherwise (for branches of type 1) 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 like in Figure 2.23. In some typical environments with repetitions (see figures 2.24 – 2.29) the (default) module head may be omitted.

sections
⋆ \section - - - *repeated for branches*
*
{Abstract} - - - *not numbered*
In this . . .

{Previous work}
A lot of . . .

⋆ \subsection - - - *repeated*

. . .

. . .

{Future work}
. . .
*
{Acknowledgements} - - - *not numbered*
. . .
sections

Figure 2.23: Repetitions as loop sketches

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

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

Figure 2.24: Tabular

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

Figure 2.27: Itemize

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

. . .
tabbing

Figure 2.25: Tabbing

enumerate
⋆ \item - - - *default*
. . .
. . .
. . .
. . .
enumerate

Figure 2.28: Enumerate

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

Figure 2.26: The bibliography

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

Figure 2.29: Description

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

Figure 2.30: Figure

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

Figure 2.31: Table

2.6.5 Captions and labels

Arrows may be used to represent captions and labels (see figures 2.30, 2.31).

2.6.6 Sketchifying

The current version of *AmadeusJ* does not support proper sketchifying of LaTeX source documents.

2.7 *AmadeusJ*: the functions

2.7.1 Basic file handling

While running *AmadeusJ*, several *current sketchy texts* may exist at a time. Each of them is connected to a separate frame and displayed in corresponding window.
Any sketchy text (in an active frame) may be exported. **Exporting** means writing the sketchy text into a file in a brows-able (*HTML*) format. There are two export modes: *plain HTML mode* and *HTML mode*. The first one ("File + Export plain-HTML") does not preserve symbols' colors and fonts, the second one ("File + Export HTML") exports symbols together with all of their attributes. Beside coding necessary information, both modes include

additional *HTML* constructs, providing some (pseudo-graphical) display of the exported sketchy text while browsed. The first mode ignores icon-tags: any scheme will be displayed as non-iconified. The second mode builds a hyper-linked *HTML* text, in which occurrences of iconified schemes are replaced by hyper-links. The real contents of iconified schemes will be displayed separately, after the sketchy text.
Importing (”File + Import”) means reading an (exported earlier) sketchy text from the file and connecting it to the active frame. The ”old” current sketchy text of the frame will be lost. For the imported file of plain *HTML* mode, symbols will be assigned (current) default colors and fonts.
Writing (”File + Write text”) is applicable for current plain sketchy text. The latter will be saved as corresponding plain text, ignoring current colors and fonts.
Reading (”File + Read text”) means reading a plain text from a file, building corresponding plain sketchy text and connecting it to the active frame. The ”old” current sketchy text of the frame will be lost.
A **new** empty sketchy text with base language ”None” may be connected (”File + New”) to the active frame. The ”old” current sketchy text of the frame will be lost.
Other file-related functions (”Import macro” and ”Print plain-TEX”) are considered in the following sections.
Remark. Prompting for safety (”File exists!”, ”Save current sketchy text?” etc.) is not implemented.

2.7.2 Editing

To set a **selection** in the current sketchy text and position the caret, mouse left button is used (see Table 2.2). Double click selects a somewhat wider area. Dragging (with left button pressed) selects either a sub-text (when inside a text or a row) or a group of branch members (when dragged over several branch members). Mouse right button clicking is used for **toggling icons**, i.e. for iconifying/deiconifying schemes. These functions are algorithmically defined in Figure 2.32.
Rather common operations ”Edit + Cut” and ”Edit + Copy” are applicable for selected items. Though, compared to pasting in plain text editors, ”Edit + Paste” is more specific: the clipboard item is inserted relatively to caret, but not necessarily at the caret. The rules for inserting, described in the next section, are followed. Some information about the kind of the clipboard

?? click count

? single:
a selection exists **?**

single scheme is selected **?**
toggles the selected scheme - - - *no matter where clicked*

a group of branch members is selected **?**
toggles all member sketches
in the selected group - - - *no matter where clicked*

- - - *selection other than scheme or group, no action*

- - - *no selection*
clicked on scheme (left border) or icon **?**
toggles the clicked item

- - - *no scheme clicked, no action*

? double:
clicked on scheme (left border) or icon **?**
toggles the clicked item and all schemes at the the same level

behaves like single click

Figure 2.32: Mouse right button click functions

Table 2.2: Mouse left button click functions

Clicked on . . .	Single click	Double click
primitive (or head) text	text row selected and caret set	primitive selected and caret set
primitive comment	comment row selected and caret set	primitive selected and caret set
scheme comment	comment row selected and caret set	scheme selected and caret set
branch line	branch selected	super-sketch selected
sketch line start	sketch selected	sketch selected
sketch line end	sketch selected	sketch selected
sketch line (inner)	nearest branch selected	sketch selected

current content is provided by "Edit + Paste?".
Deleting ("Edit + Delete") is always applicable. If caret is positioned in a comment, then if the comment is empty, the commented item (primitive, primitive head, or scheme) will be deleted; otherwise the non-empty comment will be emptied. If caret is positioned in a text part, then if the text and its comment are empty, the primitive (primitive head) will be deleted; otherwise the text will be emptied.
Toggling with "Edit + Toggle" means, first of all, changing primitive or scheme types. In particular, the current selection-group may be enclosed into a simple scheme, and a selected headless module sketch with an empty comment may be opened (see Table 2.3). Note that a single selected primitive is considered as a trivial group of branch members. A selection can be enclosed into a simple sketch also by "Edit + Enclose". It enables to enclose a single selected sketch as well.
A group of symbols selected as (part of) text or comment may be given new **color attributes** ("Edit + Toggle text color)" or set to **default color** ("Edit + Default text color").
All functions "Edit + . . ." described above are subject to **undo** operation "Edit + Undo". The redo function is not implemented.

Table 2.3: Toggling of items and groups

Current selection	"Edit + Toggle"
primitive (or head) text (part)	changes primitive (or head) type
simple scheme	changes type (0) of scheme (to 1)
loop scheme	changes type (1) of scheme (to 2)
module with head or comment	changes type (2) of scheme (to 0)
headless uncommented module	opens the module scheme
group of branch members	encloses the group into simple scheme

2.7.3 Inserting

Schemes of all types, primitives, primitive heads, arrows and some of their special forms may be inserted using corresponding functions "Insert + . . .". As a rule, an item is inserted at the possible for that item location, nearest to current caret position. This group of functions includes also means for shortening ("Insert + Arrow−−") and prolonging ("Insert + Arrow++") arrows, pointed to by caret.
All functions "Insert+ . . ." are subject to undo operation "Edit + Undo".

2.7.4 Edit-macros

A set of macro definitions can be described to facilitate editing.
The *macro definition* consists of a macro command, a macro body and an optional macro comment. The set of macro definitions is given in the last (possible single) branch of a sketchy text. This sketchy text must be activated by menu function "File + Import macro" to enable search for macros (from this imported sketchy text). **Importing macro** involves creating a new frame, importing the given file into it and notifying the macro handler. The sketchy text imported as macro will be used as active set of macro definitions for all frames (including this sketchy text as well). It may also be currently changed. It remains active even after closing its frame. To cancel the active set of macro definitions, one should import a new set of macro definitions, or replace it by an empty sketchy text ("File + Import macro" + "Cancel", for instance).
Only one set of macro definitions can be active at a time. The active set

is valid for all frames. While editing, pressing "Alt" key initiates input of macro command: all (ordinary non-space) characters typed until "Alt" is released, form the macro command. When "Alt" is released, corresponding to the command macro definition is searched in the active set; if found, the body of macro definition is inserted at current caret: first, it is copied to clipboard and, second, pasted at current caret.
There are no macros with parameters.
A particular macro definition is represented as a branch of some collection of macro definitions. Let ST be a sketchy text, which will be used to solve macros, i.e. will be imported by the menu function "File + Import macro". In such case, any sketch in the last (possible single) branch of ST will be considered as a *collection* (subset) of macro definitions. In Figure 2.33, a general structure for representing a set of 7 macro definitions is given. In this example, the set is split into 2 collections. Let b be a branch representing a macro definition. Then the comment of branch b must start with corresponding macro command, and may be suffixed by one or more spaces followed by the macro comment. The macro body (to be inserted when the macro is executed) is algorithmically defined in Figure 2.35.
A sample sketchy text structure with several collections of macro definitions is depicted in Figure 2.34. Note that macro command *sop* (in Figure 2.38) is commented (by a space), otherwise its body would be considered simply as text 'System.out.println(" " +);'.

2.7.5 Printing

The current version of *AmadeusJ* supports printing sketchy texts only via TEX (LATEX).
Activating menu item "File + Print plain-TEX" writes the current sketchy text into a file as a plain-TEX-text containing macros (from file "amadeus.tex") for drawing schemes in a style close to view *Sketches* (see Section 2.2.2). The resulting file starts with "\input amadeus.tex" and ends with "\bye" so being ready for TEX-processing.

ST - sketchy text containing macro definitions (an overview)
an arbitrary branch (optional), ignored while searching for macros
. .
macro definitions (the last branch in ST)
. . - - - *optional group of primitive heads, ignored while searching macros*
. . - - - *optional group of primitives, ignored while searching macros*
macro collection (subset) I
. . . - - - *optional group of primitive heads, ignored while searching*
branch: macro definition 1
branch: macro definition 2
branch: macro definition 3
macro collection (subset) I
. . - - - *optional group of primitives, ignored while searching for macros*
macro collection (subset) II
. . . - - - *optional group of primitive heads, ignored while searching*
branch: macro definition 4
branch: macro definition 5
branch: macro definition 6
branch: macro definition 7
macro collection (subset) II
. . - - - *optional group of primitives, ignored while searching for macros*
ST - sketchy text containing macro definitions (an overview)

Figure 2.33: Macros

▮ *any sketchy text*
a sample set of macro definitions
▮ *German letters (see Figure 2.36)*
▮ *text fragments (see Figure 2.37)*
▮ *primitives (see Figure 2.38)*
▮ *structures (see Figure 2.39)*

Figure 2.34: Some collections of macro definitions

Given: branch b – a macro definition
head of b is not empty **?**
- - - *in this case the body of branch b is insignificant.*
- - - *Hint (for developing of such macro definition): in this case, the*
- - - *primitive heads form the macro body (to be inserted into any head);*
- - - *therefore such macro definition should be assigned to base "None"*
- - - *enabling to toggle through all primitive head types*
macro body := all primitive heads of branch b;
- - - *b is headless*
body of b is empty **?**
macro body := ; - - - *empty (nothing will be inserted)*
- - - *b is headless and its body is not empty*
body of b consists of a single primitive **?**
comment of b is empty (does not contain any symbol) **?**
- - - *this is the case for inserting a symbol or group of symbols*
macro body := text of this single primitive
- - - *b is headless and its body is not empty*
- - - *and there are more than one member in body,*
- - - *or the single member is a sketch,*
- - - *or the single member is a primitive with comment*
macro body := all members of branch b

Figure 2.35: Definition of macro body

German letters
a
ä
A
Ä
o
ö
O
Ö
u
ü
U
Ü
German letters

Figure 2.36: Macros for letters

primitives
sop print line
System.out.println(" " +); - - -
os
o instanceof Sketch **?**
Sketch s = (Sketch)o;
fori i = 0, 1, . . .
⋆ int i = 0; i < ; i++
forj j = 0, 1, . . .
⋆ int j = 0; j < ; j++
primitives

Figure 2.38: Macros for primitives and primitive heads

text fragments
/ backslash in TeX
$\backslash$
imp
import java.
bz body size
.body.size()
text fragments

Figure 2.37: Macros for text fragments

structures
main
main
public static void main(String[] v)
main
ife if . . . else . . .
?
structures

Figure 2.39: Macros for structures

The printing style differs from the display view *Sketches* in 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 the print style, the sketch comment is also always duplicated at the end of the sketch line. Branches are attached to their parent-sketch lines and have no own "branch lines".
To specify the branch type, corresponding separator (type-tag) is shown in front of branch comment: "-" for a simple branch, "=" for a loop branch, and bold "**-**" for a module branch (see also Table 2.1). The separator is omitted in the case of a single headless simple branch with empty comment. So, for such loop sketch, the horizontal part of the sketch line looks like doubled. The last horizontal part of the loop sketch line is always doubled. For shorter simple and loop sketches a one-row form is used.
Primitives of type 1 are suffixed by "?" (instead of prefixing, as in the sketchy view).
It is supposed that primitive texts in the current sketchy text are already written in accord with TEX rules. If appropriate, the special TEX-preparing tool ("Tools + Prepare Tex") could be used; the tool is currently available for base language *Java*.
To obtain normal figures printed from TEX, their height must not exceed the page height. Therefore it should be taken care to apply this feature only for sufficiently short sketchy texts. If necessary, appropriate iconifying should be taken prior to "Print plain-TEX". For splitting larger sketchy texts, a simple documentation feature (described in the next section) may be helpful.

2.7.6 Documentation

The tool "File + Print plain-TEX" contains some simple automatic splitting facility, related to the first level module sketches in the current sketchy text. In the case when such module sketch, say, module m, has a comment suffixed by asterisk "*", the contents of its branches (i.e primitives and sub-sketches) is printed separately after the sketchy text. Since it is done without the enclosing scheme line of m, the contents is page-breakable. At that, those sub-sketches which were iconified in m are now deiconified and others – iconified. All the contents-group is given a heading, containing the comment of m without "*".
In particular, it suggests the following documentation technique. A larger

sketchy text should be shortened to an one-page figure using iconifying (some of) its first-level sub-sketches. After that, those of iconified sub-sketches which are desirable to be separately shown as opened (deiconified), should be enclosed into an extra module sketch with an asterisked comment. Such module may also contain non-iconified sketches – they, in turn, will be shown as icons in the separate contents-part. It should be taken care that the sub-sketches deiconified in the contents-part would be of reasonable length (i.e. not exceeding one page).

2.7.7 Other functions

In this section, functions not mentioned above are briefly explained.
"Base + ?Current": displays the base language name for the selected scheme.
"Base + Set": sets the requested base language for the selected scheme and its sub-schemes.
"Preferences + Locale": sets the requested user interface language for all frames. English and Estonian are currently available.
"Preferences + Toggle selection fill": toggles between grayed and border-lined selection styles (for all frames).
"Frame + New graphics frame": creates and opens a new graphics frame (with no sketchy text connected to it). The current version of *AmadeusJ* does not support text frames.

Conclusion and further work

In this report, present achievements in the field of sketchy modeling and programming were outlined. Altogether, the current developments in the area form a solid starting point for designing and implementing quality software products for SKM.
The general modeling concepts should be fully implemented, using, possibly, the current version of *AmadeusJ* as a prototype. First of all, the notion of hyper-symbol must be incorporated. It would result in essentially higher flexibility and modeling power of the tools. Another urgent task is to develop automated sketchifiers and textualizers which make use of formally defined sketchy mappings. It is inevitable for opening the way to really intensive and wide applications of SKM.

Bibliography

[1] J. Kiho. Sketchy programming. *Technical Report of Computing Center*, University of Tartu, no 50, 1983, pp. 52-68 (in Russian)

[2] J. Kiho. On sketchy programming technique. Proc. *The 5-th Seminar for University Teachers of Mathematics*, Oct. 12-14, 1983, Vilnius, pp. 143-145 (in Russian)

[3] J. Kiho. Sketchy programming. University of Tartu, 1984, 36 p. (in Estonian)

[4] J. Kiho. Programming of loops. University of Tartu, 1984, 70 p. (in Russian)

[5] J. Kiho. One way of improving microcomputers' software. Proc. *All-Union Conference on Individual Dialogue Systems for Microcomputers (PC): DIALOGUE-84-MICRO*, Nov. 19-22, 1984, Leningrad, "Nauka", pp. 135-137, 1984 (in Russian)

[6] J. Kiho. Sketchy representation of structures. *Technical Report of Computing Center*, University of Tartu, no 52, 1985, pp. 55-77 (in Russian)

[7] J. Kiho. Basics of informatics and computers I. University of Tartu, 1985, 32 p. (in Russian)

[8] J. Kiho. Teaching control structures, based on a common algorithmic language. Proc. *Theoretical and Applied Issues on Mathematics*, Sept. 26-27, 1985, Tartu, pp. 56-58 (in Russian)

[9] J. Kiho. Fortran-based sketchy programming. . Proc. *Theoretical and Applied Issues on Mathematics*, Sept. 26-27, 1985, Tartu (in Russian)

[10] P. Abel, J. Kiho. Sketchy programming language *SAPHIR*. . Proc. *Theoretical and Applied Issues on Mathematics*, Sept. 26-27, 1985, Tartu (in Russian)

[11] J. Kiho, M. Tisler. Instructions for sketchy programming in time-sharing system TSO. University of Tartu, 1985, 23 p. (in Estonian)

[12] J. Kiho, K. Kullman. A sketchy programming system for microcomputers. Proc. *All-Union on Programming Technology*, part II, Kiev, 1986, pp. 177-177 (in Russian)

[13] P. Abel, K. Karu, J. Kiho. Sketchy programming for microcomputers. *Technical Report of Computing Center*, University of Tartu, no 54, 1986, pp. 145-157 (in Russian)

[14] J. Kiho, Ü. Kährik. Introducing the sketchy programming technique. Proc. *The 4-th Regional Conference "Experience on applying technical tools in teaching process"*, part II, Vilnius, 1986, pp. 167-168 (in Russian)

[15] J. Kiho. Introduction to computers I. University of Tartu, 1986, (2nd ed: 1987) 56 p. (in Estonian)

[16] J. Kiho. E. Saks. Instructions for sketchy programming under TSO *DINTES*. University of Tartu, 1986, 40 p. (in Estonian)

[17] J. Kiho. E. Saks, J. Ehasalu. Developing a programming system for students' computer class and the experience of usage. Proc. *Seminar for University Teachers of Mathematics*, March 31 - April 2, 1987, Tallinn, pp. 150-151 (in Russian)

[18] J. Kiho. Sketchy Fortran. *Arvutid ja andmetöötlus*, no 3, 1988, pp. 1-6 (in Estonian)

[19] J. Kiho, E. Saks. A sketchy programming system for EC. *Technical Report of Computing Center*, University of Tartu, no 55, 1988, pp. 65-81 (in Russian)

[20] J. Kiho, K. Kullman. The sketchy language standard and a sketchy editor. *Technical Report of Computing Center*, University of Tartu, no 57, 1989, pp. 93-111 (in Russian)

[21] K. Kullman, J. Kiho. Sketchy programming technology. In *Ways of improving effectiveness of using computer technology*, Tallinn, 1989, pp. 77-80 (in Russian)

[22] J. Kiho. Introduction to computers II. University of Tartu, 1989, 104 p. (in Estonian)

[23] J. Kiho. E. Mashirina, J. Mashirin. Basics of informatics and computers II. University of Tartu, 1990, 112 p. (in Russian)

[24] J. Kiho. Elementary algorithms. University of Tartu, 1991, 144 p. (in Estonian)

[25] 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.

[26] J. Kiho, M. Muts. SKC: A C-based sketchy programming system. Proc. *Nordic Workshop on Programming Environment Research*, Tampere, Finland, Jan. 8-10, 1992, 6p.

[27] J. Kiho. For-loops. University of Tartu, 1993, 36 p. (in Estonian)

[28] J. Kiho. J. Pöial. Conditional schemes. University of Tartu, 1993, 12 p. (in Estonian)

[29] J. Kiho. Algorithms and data structures. University of Tartu, 1994, 87 p. (in Estonian)

[30] J. Kiho. Self-reducing control structures in the sketchy programming environment. Proc. *Nordic Workshop on Programming Environment Research*, Lund, Sweden, June 1-3, 1994, 333-334

[31] J. Kiho. Quasi-structured programs. Proc. *4th Symposium on Programming Languages and Software Tools*, Visegrád, Hungary, June 8-14, 1995, 67-78

[32] 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.

[33] J. Kiho. Quasi-Structured Programming Techniques. Abstracts *The 8-th Nordic Workshop on Programming Theory*, Oslo, Norway, Dec. 4-6, 1996, pp.11-11

[34] J. Kiho, A. Jalas. Retrieval system of "CIS-TARTU" data bank. *Organic Reactivity*, vol. 25, 3(91)-4(92), 1988, pp. 322–329

[35] V. Palm, A. Jalas, J. Kiho, T. Tenno. A computerized system for storage, processing and prognostication of data with orientation towards the use of correlation equations. *Organic Reactivity*, vol. 31, 1(104), 1997, pp. 111–133

[36] J. Kiho, D. Brodski, A. Podlipski. Control Structure Transformations in the Program Multiviewer. Proc. *The 5-th Symposium on Programming Languages and Software Tools*, June 7-8, 1997, Jyväskylä, Finland, pp. 61-71

[37] J. Kiho. Algorithms and data structures. (Second ed.) University of Tartu, 1997, 124 p. (in Estonian)

[38] J. Kiho. Quasi-structured programs. *Annales Universitatis Scientarum Budapestiensis de Rolando Eötvös Nominate Sectio Computatorica*, XVII, 1998, pp. 219-234

[39] J. Kiho, V. Hagu, M. Sutt. 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, pp. 221-225

[40] J. Kiho, R. Ustitch, M. Tudre. Designing computer texts with *AMADEUS*. Proc. *Fenno-Ugric Symposium on Software Technology FUSST'99*, Aug. 19-21, 1999, Sagadi, Estonia, pp. 151-162