The AGILE system[1]

Jiří Hana

Contents

[1]The work reported in this paper has been supported by the INCO-COPERNICUS Programme of the European Commission, Grant No. PL961104.

1 Introduction

The AGILE (Automatic Generation of Instructions in Languages of Eastern Europe) system is a tool for generating continuous instructional passages found in CAD-CAM manuals in Bulgarian, Czech, Russian and English. The only necessary thing is to specify the content of these passages in a language-independent content model. The system then generates various sections of the CAD-CAM manual (user's guide, quick reference, overview, etc.) in selected languages and selected styles (personal or impersonal).

This paper begins with a description of the system from the user's point of view including the creation of a simple content model. In the next chapter the general structure of the system is described. The following chapters discuss individual parts of the system – first the T-Box, then the text and sentence planners, finishing with grammars, lexicons and morphological modules. Finally, the paper presents results found during the evaluation of the system.

2 User's point of view

In this section, we describe the system from the user's point of view, leaving as much technical details as possible for the following sections. First we present an example of the text aimed by the project, then creation of a simple content model and finally the generation process and its results are described.

2.1 Sample text

As stated before, the system is suitable for generating texts found in CAD-CAM manuals.[2] A sample English text is shown in Figure 1.

The system is able to produce texts of various styles (Full Instructions, Short Instructions, Functional Description, Overview, Table of Contents) and styles (Personal or Impersonal).

2.2 Creating the text model

The author of the texts, user of the Agile system, has to provide necessary information about the instructions to be expressed by the generated manuals.

This information, the content model of the manual, has the form of an A-box (see section 4.2) – a semantic description organized in Attribute Value Matrixes (AVM's). For each described task, it is necessary to specify its Goal and Methods for achieving it. A Method contains again a list of tasks, which, however are simpler than the original one – the recursive decomposition ends when the task is simple enough to leave the methods unspecified. This organization mirrors the standard structure of describing tasks in software manuals.

The author first creates a new model and clicks the *start* button – a *procedure* with unspecified attributes (see the screenshot in Figure 3[3], or the screenshot in Figure 2). Instead of values, the slot of each attribute specifies only their types. In the Editor, obligatory attributes have red slots.

The author first clicks the *goal* slot and selects *draw* from the menu (see Figure 4). After that the model will have structure shown in Figure 5. Then s/he has to specify the *actee* (patient) of the drawing – i.e. the *line*. S/he clicks the *actee* slot and selects the *line* item from the menu.

[2]The system (Lexicons and Domain model) is prepared for generating AutoDesk AutoCAD manuals, however the effort of modification for other CAD/CAM products should be minimal.

[3]The depicted interface is in English, however the user can select between Bulgarian, Czech, English and Russian. All menus and dialogs, but also names, types and values of attributes appear in the selected language. The language of the interface is not dependent on the language(s) selected for generation.

To create a multiline style

1. First open the multiline Styles dialog box using one of these methods:
    - **Windows:** From the Object Properties toolbar or the Data menu, choose Multiline Style.
    - **Dos and Unix:** From the Data menu, choose Multiline Style
2. Choose Element Properties to add elements to the style.
3. In the Element Properties dialog box next to Offset, enter the offset of the line element.
4. Select Add to add the element.
5. Select an element.
6. Choose Color. Then select the element's color from the Select Color dialog box.
7. Select an element.
8. Choose Linetype. Then select the element's linetype from the Select Linetype dialog box.
9. Choose OK to save the multiline element style and exit the Element Properties dialog box.

Figure 1: Sample text in English

Now the author has completed the specification of the *goal* of the task and can proceed with specification of the *methods*. Each *method* describes one alternative way for achieving the *goal* (e.g. by mouse or keyboard). In our example, there is only one *method*: *defining its start and end points*. Therefore, the list of methods will contain only one item.

The author first clicks the *methods* slot – it will be filled by a list of methods (menu does not appear, because there is nothing to select) with one method prefilled.

The author has to specify steps of the method – first step will model *define its start point*, the second one will model *define its end point*. The important thing in this two tasks is that the *owner* of the start and end points is the same *line* as the *line* in the *goal* of the top-level task. Therefore instead of inserting new instances of the *line* concept, the author has to paste a link to the *line* used in the *goal* of the top-level task. In the editor, the coindexed instances have the same identifier (the funny letter + number following the name of the concept).

Finally the model will look as in Figure 6.

2.3 Generation

When modeling, let's say, a chapter in a CAD/CAM manual, it is necessary to build similar (but most likely more complex) models for each described task – e.g. opening a drawing, creating a drawing and saving a drawing. Once these models are completed, the system can generate the

$$\begin{bmatrix} \text{procedure} & \\ \text{goal} & \textit{action} \\ \text{methods} & \textit{list of methods} \\ \text{side-effect} & \textit{event} \end{bmatrix}$$

Figure 2: An empty content model

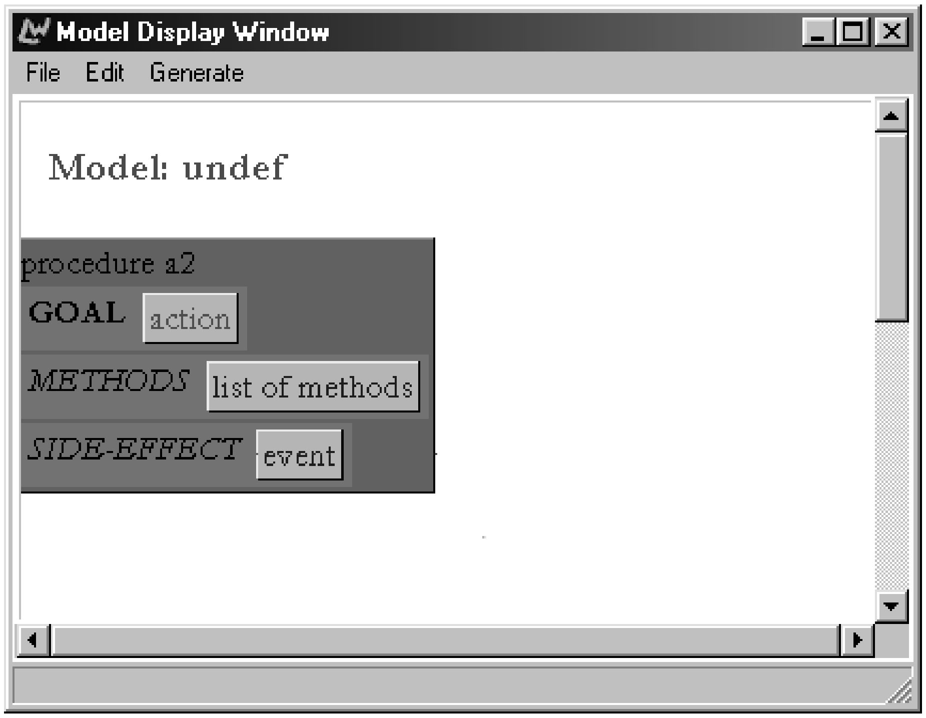


Figure 3: The editor with an empty content model

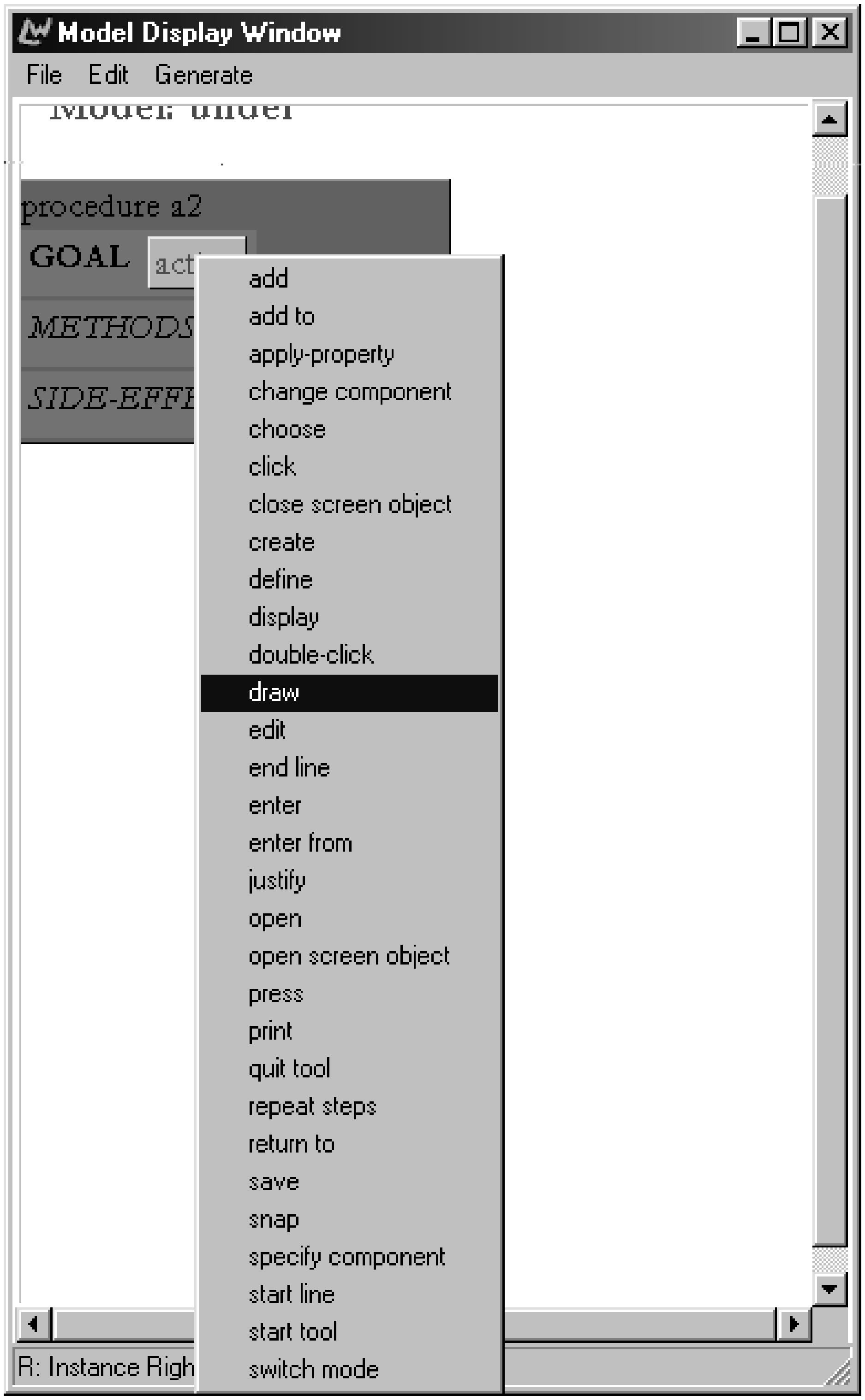


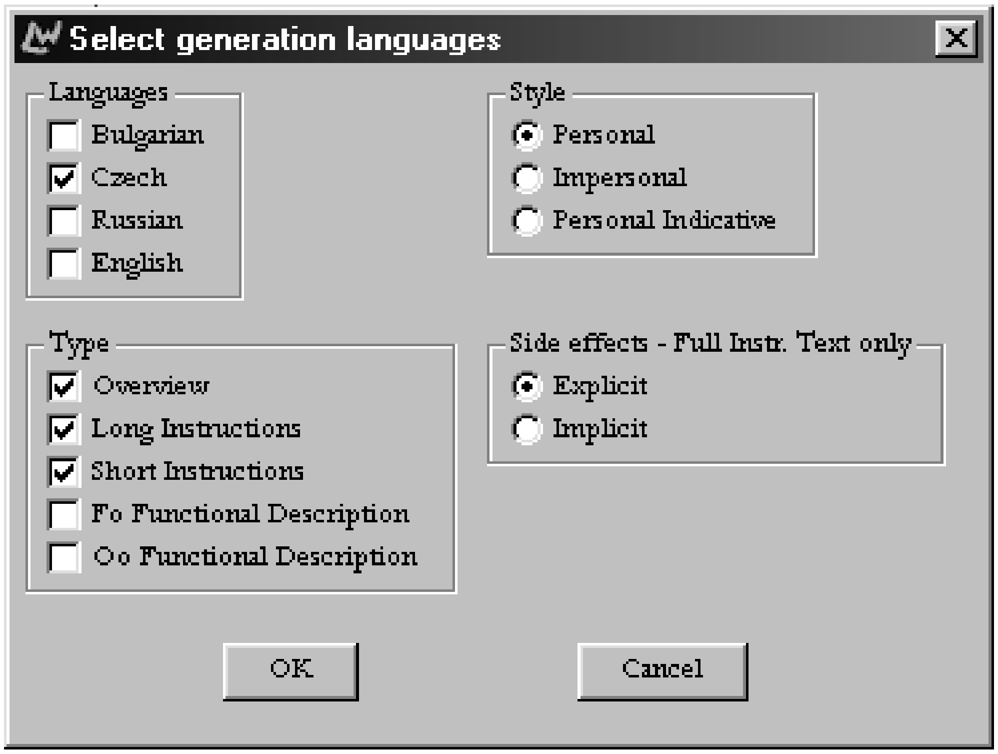
Figure 4: Filling the goal

$$\left[\begin{array}{ll}
\text{procedure} & \\
\text{goal} & \left[\begin{array}{ll} \text{draw} & \\ \text{actee} & \textit{graphical object} \end{array}\right] \\
\text{methods} & \textit{list of methods} \\
\text{side-effect} & \textit{event}
\end{array}\right]$$

Figure 5: The model with the filled Goal

$$\left[\begin{array}{ll}
\text{procedure} & \\
\text{goal} & \left[\begin{array}{ll} \text{draw} & \\ \text{actee} & \text{line } \boxed{1} \end{array}\right] \\
\text{methods} & \left[\begin{array}{l}
\text{list of methods} \\
\left[\begin{array}{ll}
\text{method} & \\
\text{precondition} & \textit{procedure} \\
\text{steps} & \left[\begin{array}{l}
\text{steps} \\
\left[\begin{array}{ll}
\text{procedure} & \\
\text{goal} & \left[\begin{array}{ll} \text{define} & \\ \text{actee} & \left[\begin{array}{ll} \text{start point} & \\ \text{owner} & \text{line } \boxed{1} \end{array}\right] \\ \text{location} & \textit{data object} \end{array}\right] \\
\text{methods} & \textit{list of methods} \\
\text{side-effect} & \textit{event}
\end{array}\right] \\
\left[\begin{array}{ll}
\text{procedure} & \\
\text{goal} & \left[\begin{array}{ll} \text{define} & \\ \text{actee} & \left[\begin{array}{ll} \text{end point} & \\ \text{owner} & \text{line } \boxed{1} \end{array}\right] \\ \text{location} & \textit{data object} \end{array}\right] \\
\text{methods} & \textit{list of methods} \\
\text{side-effect} & \textit{event}
\end{array}\right]
\end{array}\right]
\end{array}\right]
\end{array}\right] \\
\text{side-effect} & \textit{event}
\end{array}\right]$$

Figure 6: The final sample mode


Select generation languages

Languages
- [ ] Bulgarian
- [✓] Czech
- [ ] Russian
- [ ] English

Style
- [●] Personal
- [ ] Impersonal
- [ ] Personal Indicative

Type
- [✓] Overview
- [✓] Long Instructions
- [✓] Short Instructions
- [ ] Fo Functional Description
- [ ] Oo Functional Description

Side effects - Full Instr. Text only
- [●] Explicit
- [ ] Implicit

Figure 7: The Select generation preferences dialog box

texts. Using the dialog in Figure 7 the author can specify desired languages, types[4] and styles of the generated text.

The result is presented as a HTML text in an Internet browser. The output corresponding to our example is displayed in Figure 8 for Czech and in Figure 9 for Bulgarian.

3 General structure of the system

The structure of the system can be found in Figure 10.

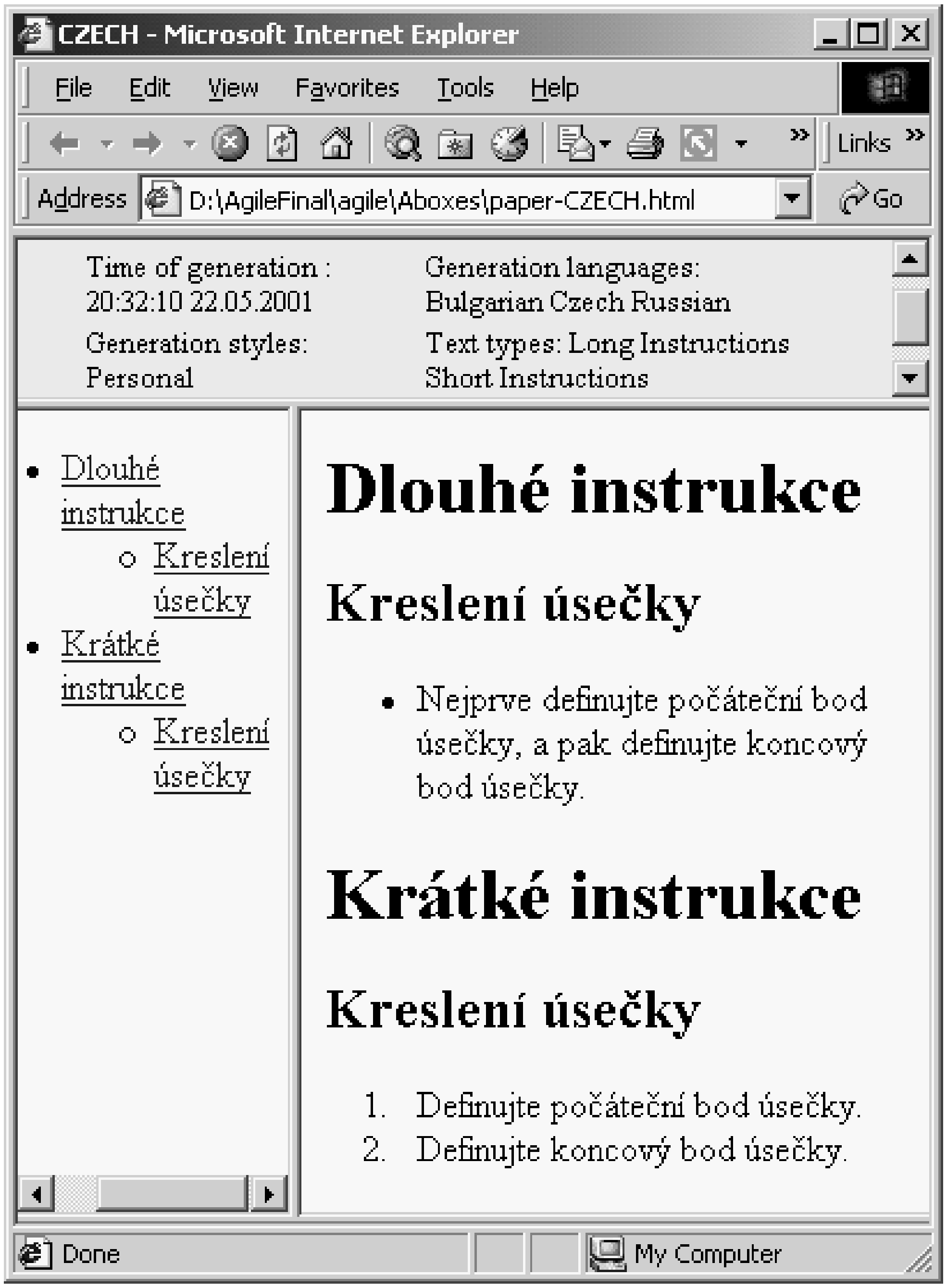
First the A-box (content model) is specified in the content model editor. Then the text plan gets the A-box an creates a corresponding text plan, which structures the specified text. On the basis of the text plan, the sentence planner produces sentence plans for individual sentences. These SPLs are then passed to the grammar, which finally produces the text. The text planner, the sentence planner and grammars are implemented in the KPML environment ([1]. T-box is consulted by all these three modules.

4 T-Box & A-Box

4.1 T-Box

The terminology-box or T-box (see [16], [17]) specifies types of objects that can occur in the worlds described by the Agile texts. The T-box is similar to a type system in typed programming

[4]The Table of contents is generated automatically, an Overview is available only for a set of tasks, not for a single one


CZECH - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Links
Address D:\AgileFinal\agile\Aboxes\paper-CZECH.html
Go
Done
My Computer

Time of generation : 20:32:10 22.05.2001
Generation languages: Bulgarian Czech Russian
Generation styles: Personal
Text types: Long Instructions Short Instructions

- Dlouhé instrukce
  - Kreslení úsečky
- Krátké instrukce
  - Kreslení úsečky

Dlouhé instrukce

Kreslení úsečky

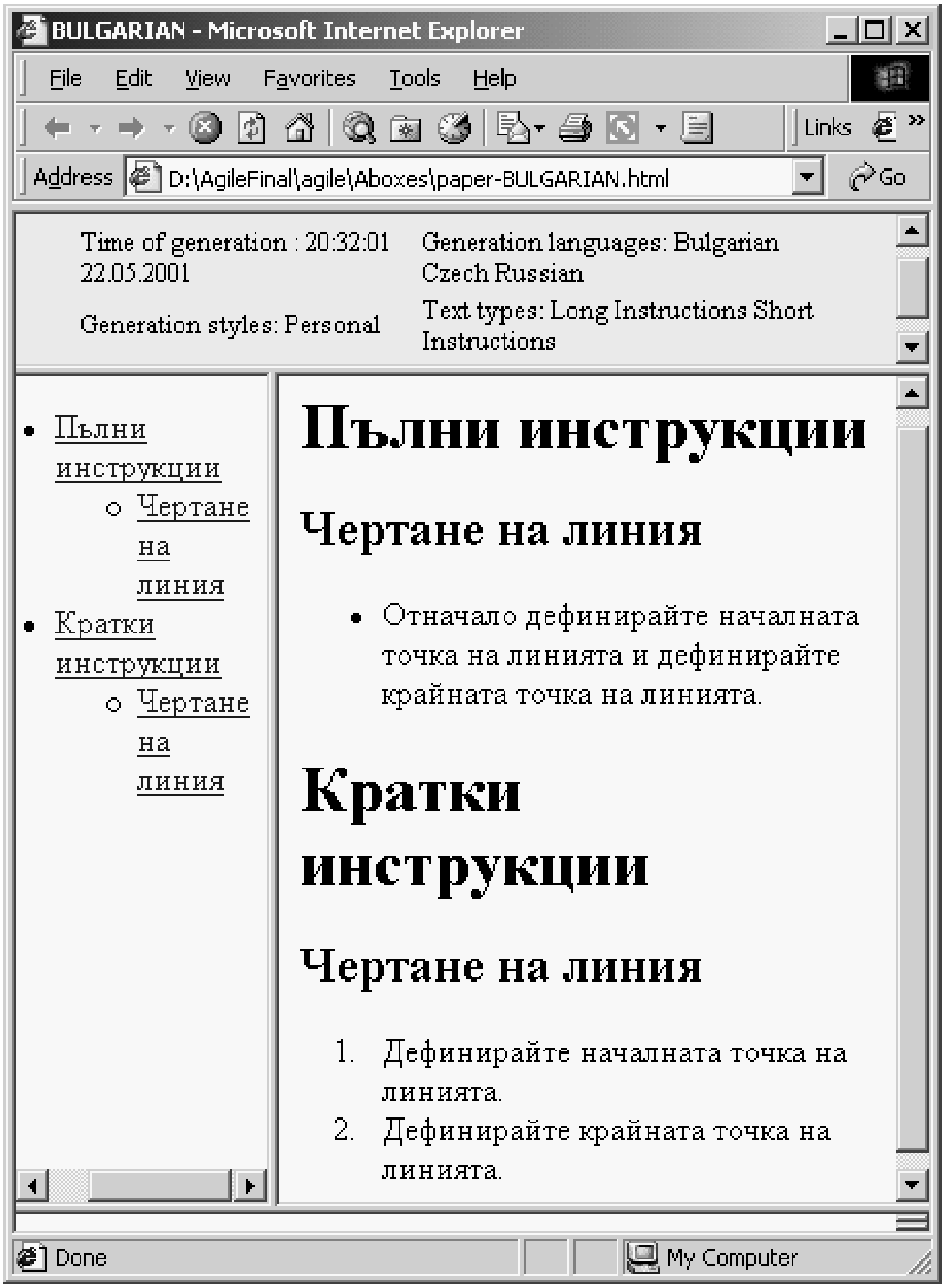
- Nejprve definujte počáteční bod úsečky, a pak definujte koncový bod úsečky.

Krátké instrukce

Kreslení úsečky

1. Definujte počáteční bod úsečky.
2. Definujte koncový bod úsečky.

Figure 8: Generated text of the sample model in Czech


BULGARIAN - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Links
Address D:\AgileFinal\agile\Aboxes\paper-BULGARIAN.html
Go
Done
My Computer

Time of generation : 20:32:01 22.05.2001

Generation styles: Personal

Generation languages: Bulgarian Czech Russian

Text types: Long Instructions Short Instructions

- Пълни инструкции
  - Чертане на линия
- Кратки инструкции
  - Чертане на линия

Пълни инструкции

Чертане на линия

- Отначало дефинирайте началната точка на линията и дефинирайте крайната точка на линията.

Кратки инструкции

Чертане на линия

1. Дефинирайте началната точка на линията.
2. Дефинирайте крайната точка на линията.

Figure 9: Generated text of the sample model in Bulgarian

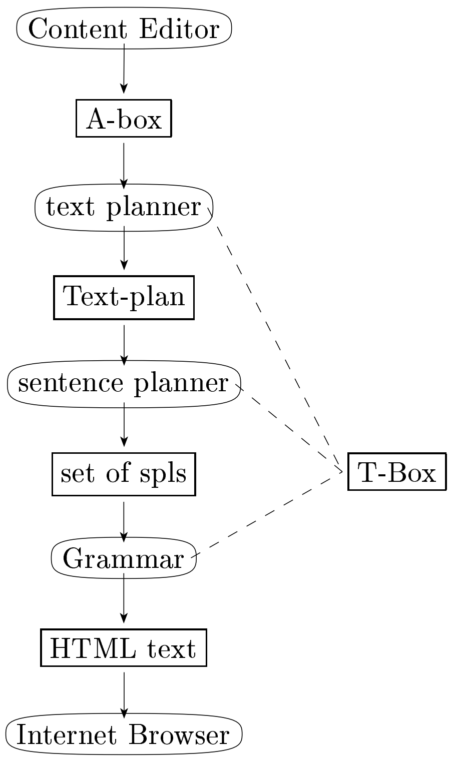


Figure 10: The structure of the system

languages like C++ or Java or to a signature in HPSG. It is a set of concepts organized in a hierarchy by the "superconcept" relation. A concept can have one or more superconcepts. Each concept defines the attributes[5] of its instances – relations to other objects. Some of the attributes are obligatory, some optional.

The Agile T-box has two parts:

- Penman Upper Model (see [2]) – general part, that specifies and classifies abstract notions like "quality", "process", etc., which are linked to natural language syntax (e.g. "quality" to the adjective, "process" to the verb).
- Domain model – specific part, that specifies the concepts for modelling CAD/CAM applications like "multiline", "command line", etc. The objects are classified by various criteria (whether they can be displayed on the screen, whether they are components of other objects, etc.). The Domain model is linked to the upper model (all domain model concepts are transitively subconcepts of some upper model concepts). This linkage specifies constraints on their linguistic realization.

A fragment of the Agile T-box is given in Figure 11.

[5]Explicitly or by inheritance from superconcepts

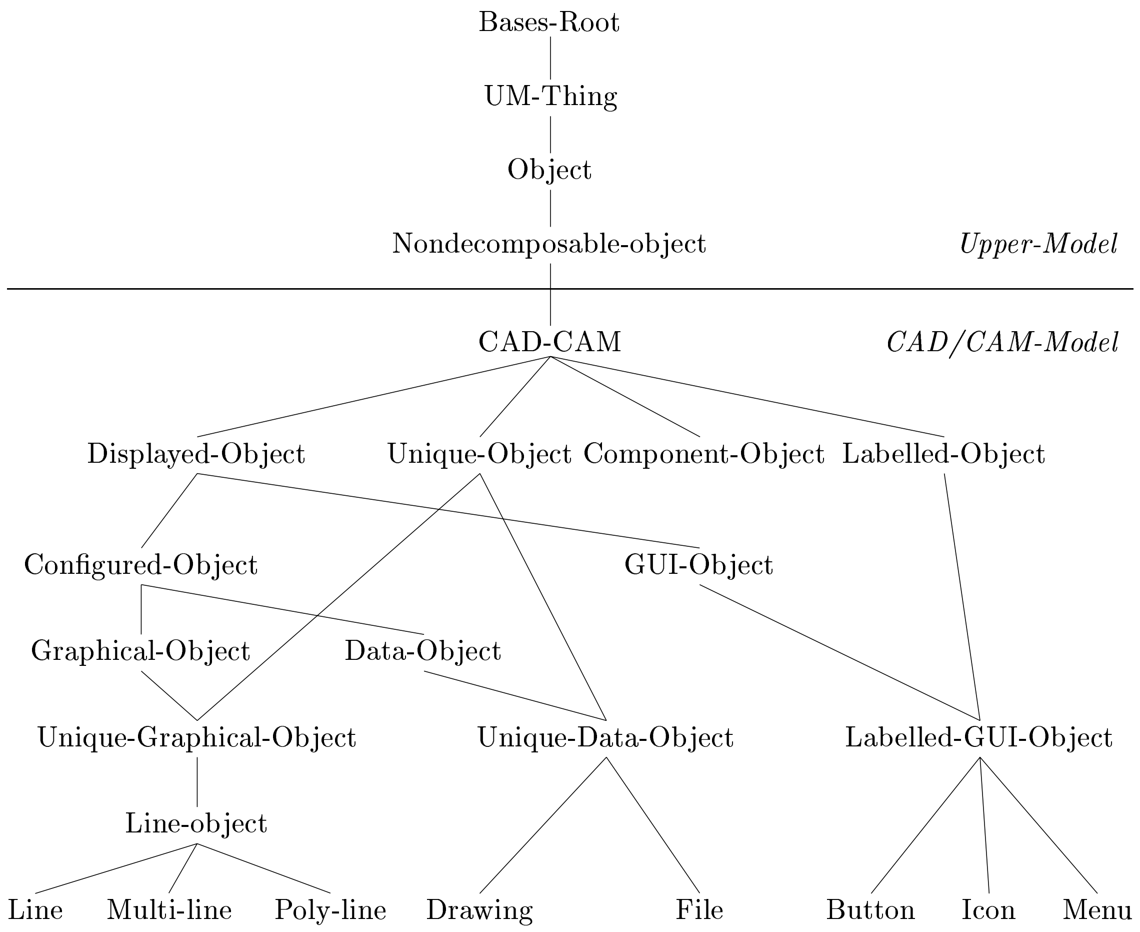


Figure 11: Fragment of the Agile T-box

4.2 A-box

A structure using instances of concepts from the T-box is called A-box (Assertion-box). It is very similar to a feature structure in HPSG.[6]

In Agile we use A-boxes for modeling instructional texts found in CAD/CAM manuals. Using Agile interface to build an A-box modeling a simple instruction was described in the section 2.2. Such an A-box has to be an instance of the concept Procedure (or set/list of Procedures) – see Figure 12.

A procedure specifies its goal, methods (i.e. alternative ways) how to reach the goal and side-effect of achieving the goal (e.g. displaying a message). Trivial procedures do not need to

[6]A HPSG feature structure is a directed graph with labeled nodes and edges. The labels on its nodes represent types (concepts) from the signature (T-box), the labels on its edges represent names of attributes.

However, in addition to the signature, a HPSG feature structure has also to satisfy a set of other constraints; there is nothing like that in Agile. They are also differently used. In HPSG, feature structures are used as universal data structures modeling both semantics and phonology of the text. For the aim of generation, you (simply put) specify the part of the feature structure containing the semantics of the generated text. Then all feature structures containing this fragment and satisfying the grammar are found – the input is a partially specified feature structure, the result is a set of fully specified feature structures.

In Agile, an A-box is used only for specifying the semantics of the generated text, other levels of description use different data structures. Therefore, when an A-box inputs the generation process, it is fully specified.

Concepts Procedure and Method informally

concept Procedure subconceptOf Instruction-Scheme
  goal :: User-Action
  methods^{opt} :: list(Method)
  side-effect^{opt} :: User-Event

concept Method subconceptOf Instruction-Scheme
  constraint^{opt} :: Operating-System
  precondition^{opt} :: Procedure
  substeps :: list(Procedure)

Real code of concepts Procedure and Method

```
(define-concept PROCEDURE (INSTRUCTION-SCHEME)
    ((GOAL :type USER-ACTION)
     (METHODS :type METHOD-LIST :optional T)
     (SIDE-EFFECT :type USER-EVENT :optional T)))

(define-concept METHOD* (INSTRUCTION-SCHEME)
    ((CONSTRAINT :type OPERATING-SYSTEM :optional T)
     (PRECONDITION :type PROCEDURE :optional T)
     (SUBSTEPS :type PROCEDURE-LIST)))
```

Figure 12: Concepts Procedure and Method

contain methods for their achievement. In fact, every nontrivial procedure has to be recursively decomposed in such trivial ones. A procedure does not simply contain a list of steps (simpler procedures), instead it specifies a list of methods each describing one alternative way (e.g. using keyboard, menu or toolbar, or different methods for different operating systems).

5 Text and sentence planners

The text structuring module (TSM, see [7] or project reports [12], [8], [4] and [9]) has two parts: a text planner and a sentence planner.

The text planner gets an A-box (content model) as input and yields a text plan that structures the specified text. The text plan consists of text plan elements that are related to the parts of the A-box they plan. On the basis of the text plan, the sentence planner produces sentence plans (SPL expressions, see 6.1) for individual sentences. These SPLs are then passed to the grammar.

5.1 Text-planner

A text plan structures the supplied A-box. It is a tree whose leaves point to individual parts of the A-box. It also specifies discourse relations that hold between these parts. A sample text plan is depicted in Figure 13.

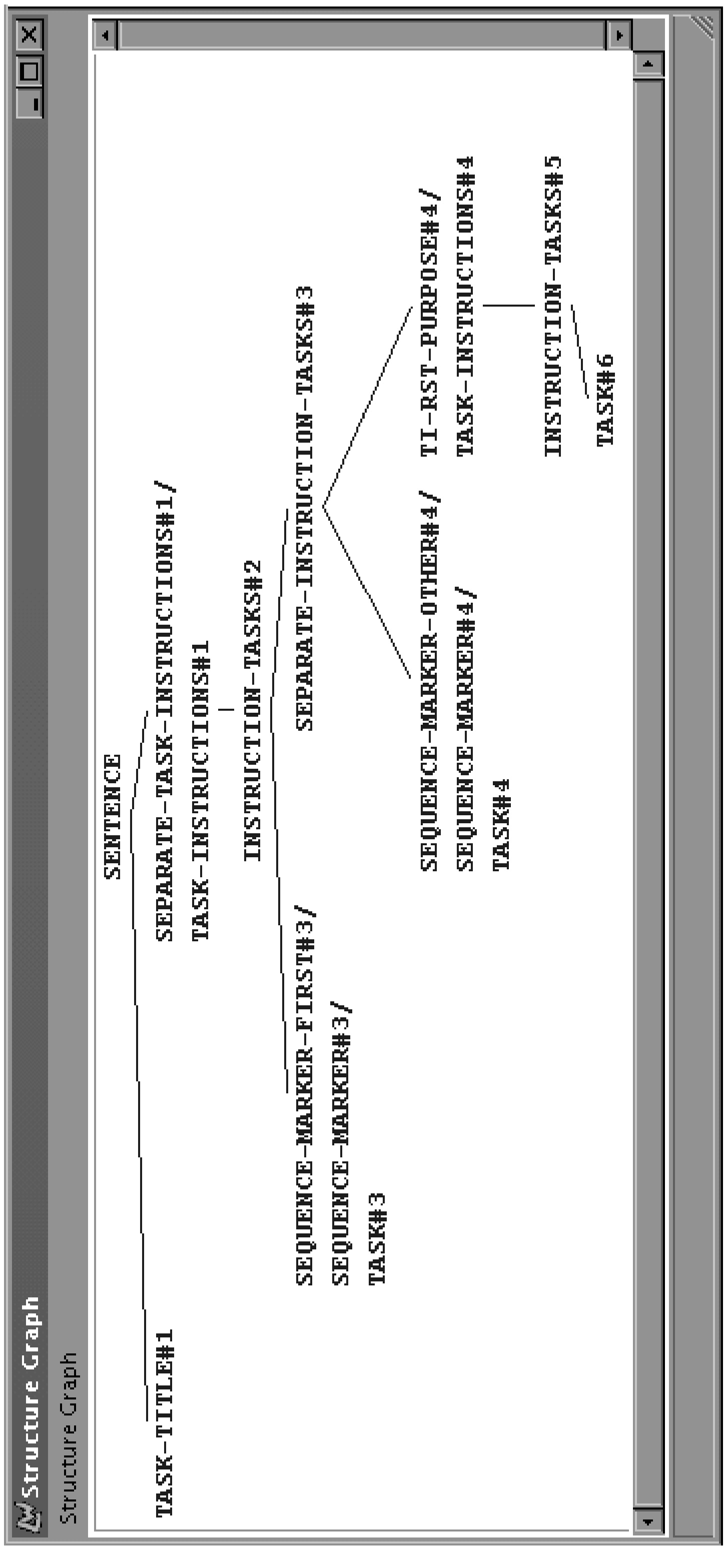


Figure 13: A sample text plan

The sentence plan is formed by the so called *text structure elements.* There is a simple mapping between these elements and the configurational concepts[7] used in the content model. The realization of text structure elements is driven by *text templates.* By constraining linguistic means of realization of the text structure element, they determine the style of the corresponding text.

For example the heading has various form in various languages:

En:	*To draw a polyline.*	(1)

Bg:	*Chertaene*	*na*	*polilinija*	(2)
	Drawing-Nominal	of	Polyline-Indef	

Cz:	*Kreslení*	*křivky*	(3)
	Drawing-Nominal	Polyline-Gen	

Ru:	*Shtoby*	*narisovat'*	*poliliniju*	(4)
	in-order	draw-Inf	Polyline-Acc	

Therefore the text template for Bulgarian and Czech has to specify that the heading will be realized as a nominal group, whereas the English and Russian ones will require the use of the infinitive construction. Text templates also constrain the layout by specifying HTML tags (e.g. heading or numbered list).

The text planner may also impose other constraints specified by the user of the system – for example whether the side effects of actions are realized or not.

In addition to the described structure consisting of text structure elements, it also distinguishes some discourse relation. Used discourse relations are motivated by the Rhetorical Structure Theory (RST, see [14]). These relations are part of the used Upper Model. Examples of such a relation are Manner or Purpose.

Along with a text plan the text planner also builds a discourse model (see [11] or [10]). This discourse model is used by the sentence planner to determine whether an item was mentioned in the preceding context and should therefore be contextually bound, or whether it should be contextually nonbound.

5.2 Sentence-planner

The sentence planner receives a tree-like text plan with small pieces of the A-box in its leaves. Each of these pieces of the A-box essentially corresponds to a clause. The sentence planner first translates these pieces of the A-box into SPL expressions, each specifying a clauses and than aggregates them into bigger SPLs, each specifying a sentence. When deciding whether and how to combine clauses into sentences, the sentence planner is driven by relations imposed by the text planner (discourse relations, coordination, sequences). For an example of a non-aggregated an aggregated text, see Figure 14.

The created SPL's also contain constrains determining their contextually appropriate information structure. This reflects the discourse relations between clauses and the discourse model built by the text plan. The generated sentence reflects these constraints by word order and by referring expressions (pronouns and articles).

6 Grammar

The grammars of the Agile system are implemented in the KPML enviroment ([1]). Each grammar gets a set of SPLs from the sentence planner and yields corresponding sentences. In

[7]This means a procedure and a method and lists of them – see 4.2.

To save a document under a different name

1. Select the Save As command from the File menu.
2. Enter the file name in the File name box.
3. Click the OK button.

To save a document under a different name, select the Save command from the File menu, enter the file name in the File name box, and click the OK button.

Figure 14: Aggregation example

this section we first describe some general issues of the KPML grammars, then mention some of the Agile specifics. After that, we discuss lexicons and morphology modules. The section closes with a simple case study (implementation of a quantified nominal groups).

6.1 A Grammar in KPML

A KPML grammar gets an SPL expression (see Figure 15 or Figures 19, as input and yields a syntactic structure as output (see Figure 20). Usually only the list of terminals of this structure is considered as output.

A grammar in the KPML system is organized in a network. The syntactic tree of the generated sentence is recursively built by successive traversals through it – for each[8] inserted nonterminal one traversal through the grammar is performed.

The network is a set of systems which are for organization reasons grouped into regions. We will now describe structure of systems on a sample system – see Figure 16. For the rest of this paper we use a simpler informal notation for describing grammar systems[9] – see Figure 17. This system asks SPL if polite or personal addressing should be used, and inflectifies the verb appropriately.

Each system has a unique name (`Indicative-Politeness-Type`). The string `Addressee--Subject` in input conditions means that the system `Indicative-Politeness-Type` is entered after passing the grammatical feature (see the next paragraph) `Addressee-Subject`, present in another system (`Indicative-Interactant-Subject`). The input conditions can be more complicated – features can be connected by conjunction and disjunction (negation is not available). For example the input condition `(NumerativableCases and More-Than-Four)` in the system `Numerative` means that the system is entered only if during the traversal of the grammar both features `NumerativableCases` and `More-Than-Four` are passed. During each traversal of the grammar every system is entered at most once depending on its input conditions.

Every system contains at least one grammatical feature (`Polite-Indicative`, `Personal--Indicative`). If there is more than one feature, KPML has to decide which one is entered. There are two main ways how to do it:

1. Run the chooser belonging to the system

[8]If not conflated with another one – see the `conflate` operator below

[9]The Name of the chooser is omitted because there is a convention that it is simply the name of the system affixed by "`-chooser`"

```
(EXAMPLE
    :NAME             SAMPLE-1
    :GENERATEDTFORM  "Uživatel nakreslí dvě úsečky."
    :TARGETFORM      "Uživatel nakreslí dvě úsečky."
    :GLOSS (
        (:ENGLISH "The user draws two lines.")
        (:LIT "user-IS3 draw-S3 two-FP4 line-FP4") )
    :LOGICALFORM
        (S / DM::DRAW
            :ACTOR
                (AR / DM::USER
                    :CONTEXTUAL-BOUNDNESS YES
                    :IDENTIFIABILITY-Q DM::IDENTIFIABLE)
            :ACTEE
                (AE / DM::LINE
                    :QUANTITY 2
                    :CONTEXTUAL-BOUNDNESS NO))
    :SET-NAME SAMPLES
)
```

Figure 15: A sample SPL

2. Preselect specific features during the traverse through the grammar when the parent of the current node was generated. In such a case the preselection has higher priority over the chooser.

Simply put, a chooser is a decision tree with names of features in its leaves and inquiries in the non-terminal nodes. Inquiries can contain any Lisp code and are used to consult T-Box, SPL, lexicon, etc. The inquiries can also modify the syntactic structure – for example: find an appropriate lexical entry for a terminal node.

A feature can contain various operators that are used to constrain the generated tree:

- `insert` *Node* – inserts new node *Node* under the current node
- `conflate` $Node_1$ $Node_2$ – conflates (coindexes) two nodes (daughters of the current node)
- `preselect` *Node Feature* – when generating *Node*, the grammatical *Feature* will be entered
- `inflectify` *Node Form* – *Form*s are collected for *Node* and are available to the morphology afterwards (see 6.4)
- `classify` *Node LexicalFeature* – restricts lexical realization of *Node* to the lexicon items having *LexicalFeature* among it's features (see 6.3)
- `out-classify` *Node LexicalFeature* – opposite to classify
- `lexify` *Node Lemma* – *Node* will be lexically realized as *Lemma* (used especially for functional word – prepositions, conjunctions, etc.)

```
(SYSTEM
    :NAME    Indicative-Politeness-Type
    :INPUTS  Addressee-Subject
    :OUTPUTS
    (
        (0.5 Polite-Indicative
            (INFLECTIFY Finite Number-Pl-Form))
        (0.5 Personal-Indicative
            (INFLECTIFY Finite Number-Sg-Form))
    )
    :CHOOSER Indicative-Politeness-Type-Chooser
    :REGION  Mood
)
```

Figure 16: System Addressee-Subject

```
Indicative-Politeness-Type (Addressee-Subject)
    [Polite-Indicative]
        Inflectify Finite Number-Pl-Form
    [Personal-Indicative]
        Inflectify Finite Number-Sg-Form
```

Figure 17: System Addressee-Subject in the informal notation

- ordering operators – various operators used to order daughters of the current node

The presented operators (esp. preselect) allow information to be passed only in the top-down direction. Moreover, for example, for the subject-verb agreement in Czech it is necessary to pass information about the gender from the subject to the finite verb – its parent. However, when it is possible to restrict the verb, the gender of the subject is not known, because the pass through the grammar for subject did not occur, therefore the subject was not lexicalized and the (grammatical) gender of the subject cannot be determined. When the subject is lexicalized and the gender can be inspected, it is impossible to restrict the verb.

Therefore during the development of the project an additional `agreement` operator was implemented. Using this operator it is possible to connect a finite verb with its subject and to ensure that necessary inflection of the verb occurs when the subject enters specified grammatical features.

6.2 Grammars in Agile

We needed grammars for all the three target languages (plus English). As a basis we used a large scale English grammar (Nigel: see [13]). The modifications of this grammar were driven by priorities determined by a corpus-based contrastive analysis of original texts in target languages performed in the early stage of the project.

For each phenomenon determined by the corpus analysis (transitivity, agreement, word-order, quatification, etc.), we did:

1. Evaluation of the current state of the grammar

2. Grammar specification – at least covering the corpus, but as general as possible
3. Grammar implementation for one language
4. Adaptation to other languages

The multilinguality of the resources has been one of the main goals of the process, taking advantage of the similarities between the typologically similar languages. This approach significantly increased the time of development. We expected to find many common parts in the grammars of the three Slavic languages, but were surprised by the amount of features shared between them and the English one.

6.3 Lexicon

SPL formula contains a specification of concepts. Language resources contain a list of lexical items for each concept. If there is more than one lexical item, grammar can impose further constraints to select between them.

That is very useful, because the morphological modules (see section 6.4) contain derived words as single entries. Thus aspectual pairs, deverbative nouns and deverbative adjectives are listed separately. For example, in the Czech grammar, the concept DM::draw is annotated with four lexical items: the perfective verb (nakreslit), the imperfective verb (kreslit), the perfective deverbative noun (nakreslení) and the imperfective deverbative noun (kreslení). If the grammar claims a noun and imperfective aspect, 'kreslení' is used, if it wants a verb and imperfective aspect, 'kreslit' is used.

This is ensured by the following:

1. The cadcam-dm-LEX-annotations-cz.annot file, the file linking domain concepts to Czech lexical items, includes:[10]:

   ```
   (annotate-concept DM::draw :lex-items
       (:Czech kreslit nakreslit kresleni2 nakresleni2))
   ```

2. In the lexicon there are, inter alia, the following four lexical items:

   ```
   (LEXICAL-ITEM
       :NAME nakresleni2
       :SPELLING "nakreslení"
       :FEATURES (noun common-noun nominalization-noun
           countable neuter perfective)
   )

   (LEXICAL-ITEM
       :NAME nakreslit
       :SPELLING "nakreslit"
       :FEATURES (verb do-verb effective-verb disposal-verb
           transitive perfective)
   )
   ```

[10]Because internally some components of the system require ASCII letters we used the so called Ruslan notation: 2 means the accent aigu, 3 means the hacek (inverted circumflex): a2 = á, c3 = č.

```
(LEXICAL-ITEM
    :NAME kresleni2
    :SPELLING "kreslení"
    :FEATURES (noun nominalization-noun common-noun
        countable neuter imperfective)
)

(LEXICAL-ITEM
    :NAME kreslit
    :SPELLING "kreslit"
    :FEATURES (verb do-verb effective-verb disposal-verb
        transitive imperfective)
)
```

3. The system responsible for selecting aspect:

```
Aspect (Independent-Clause And (Not-Conation Or Infinitive-Conation))
    [Perfective]
        Classify Process Perfective
    [Imperfective]
        Classify Process Imperfective
```

4. The systems responsible for nominalization or verb.

6.4 Morphology

Bulgarian, Czech and Russian are languages with a relatively rich morphology (especially the latter two). To reuse existing resources, all three languages employ an external morphological module. We will describe the Czech module and mention briefly the Bulgarian and Russian ones afterwards.

As a core for the Czech morphological module, we have used morphological generator written by Jan Hajič ([3]). It has a very large morphological database containing more than 800k of lemmas producing more than 15M of word forms. For convenience and copyright reasons, we used a smaller database containing only words occurring in the generated texts. However, technically it is possible to use the big database just by replacing one single file.

Parts of the morphological module are lisp interface and two dlls. A scheme of the interaction of the parts is shown in Figure 18.

The morphgen.lisp module contains two parts:

- Mapping of the AGILE morphological features to the so-called positional tag. Positional tag is a string of 15 characters where each character corresponds to one morphological category (the unused categories are substituted by '–'). The function goes through the list of morphological features obtained from the KPML system (added by the `inflectify` operator – see 6.1) and tests one morphological category after another. If the value for the category is present in the list, the corresponding position in the tag is set to corresponding value, otherwise it is set to '–'.

- Interface to the C functions in morph.dll: for this purpose the Harlequin fli (Foreign language interface) library is used.

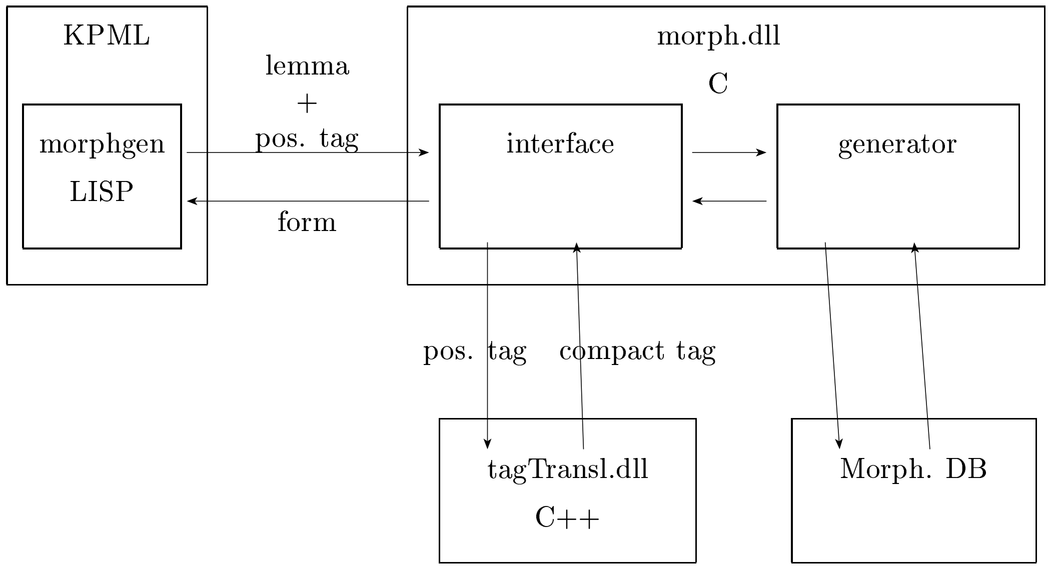


Figure 18: Czech Morphological module

The module morph.dll encapsulates the morphological generator. It receives the request from the morphgen module consisting of a lemma (base form) and a positional tag. It asks the tagTranslation.dll module to translate the positional tag to the so-called compact tag and then sends a lemma and a compact tag to the generator, which returns the generated form. If the form is not returned by the generator, the tagTranslation.dll is asked for a less specific tag and tries it once again[11]

The Bulgarian grammar reused an existing morphological module written in Object Pascal. The interface at the KPML side is a modification of the Czech one. The Russian grammar reuses existing module written in Lisp.

6.5 Example – Cardinal Numerals

6.5.1 Analysis

There are several anomalies in the declension of counted objects following a numeral in Czech and Russian.

In Czech, when the numeral is above four and the whole group should be in nominative or accusative, only numeral is in that case but the counted object is in genitive. The other cases are regular.

Dva	*body*	*zmizely.*
Two-INom	points-IPlNom	disappeared-IPl3.

‘Two points disappeared.’ (5)

Pět	*bodů*	*zmizelo.*
Five-Nom	points-IPlGen	disappeared-NSg3.

[11]This is necessary because the generator sometimes uses underspecified tags but the tags from Agile Czech grammar are fully specified. For example for the past tense form “dělala” (done) the generator uses a tag with the meaning it is either feminine singular or neuter plural (compact tag `VRQXA` corresponding to positional tag `VpQW---XR-AA---`, but our grammar uses for “dělala” two distinctive tags. (`VpFS---XR-AA---` or `VpNP---XR-AA---`, 3rd position is gender – Q = {F, N }, 4th is number – W = {S, P})

‘Five points disappeared.’ (6)

For complex numerals higher than four but with the last component one – four, there are two possibilities:

Dvacet	*dva*	*bodů*	*zmizelo.*
Twenty-Nom	two-INom	points-IPlGen	disappeared-NSg3.

Dvacet	*dva*	*body*	*zmizely.*
Twenty-Nom	two-INom	points-IPlGen	disappeared-NSg3.

‘Twenty two points disappeared.’ (7)

There are other irregular points in this subject, however they are not important for the Agile project; for more information see [6].

In Russian, the distinction above-four versus below-five and nominative-accusative versus other cases is also important. If the nominal group is in nominative or accusative the counted object is in genitive; if the numeral (or the last part in a complex numeral) is two, three or four, the counted object is in singular, otherwise it is in plural. In other cases the behavior is regular. Similarly to Czech, there are other exceptions (animate vs. inanimate) that are not important for the Agile project. To summarize:

```
if (nominal group in nominative or accusative)
    counted object in genitive;
    if (last part of the numeral is 2, 3 or 4)
        counted object in singular,
    else
        counted object in plural.
else
    normal behaviour
```

Dve	*tochki*	*izchezli.*
Two-FNom	points-FSgGen	disappeared-Pl3.

‘Two points disappeared.’ (8)

Pjat’	*tochek*	*izchezlo.*
Five-Nom	points-FPlGen	disappeared-NSg3.

‘Five points disappeared.’ (9)

6.5.2 Implementation

For space reasons we describe only the implementation for Czech. The implementation for Russian is analogous (see [19])

Before implementing these phenomena, the system inflectifying *Thing* for case has the following form:

```
Thing-Case (Nominal-Group)
    [Thing-Case-Nom] inflectify Thing Case-Nom-Form
    [Thing-Case-Gen] inflectify Thing Case-Gen-Form
    [Thing-Case-Dat] inflectify Thing Case-Dat-Form
    [Thing-Case-Acc] inflectify Thing Case-Acc-Form
    [Thing-Case-Voc] inflectify Thing Case-Voc-Form
    [Thing-Case-Loc] inflectify Thing Case-Loc-Form
    [Thing-Case-Ins] inflectify Thing Case-Ins-Form
```

Selection between features is driven by preselection from higher rank (e.g. by inspecting the valency frame of the verb). There can be statement `preselect Subject Thing-Case-Nom`.

However, in some cases we want a different behavior. The inflection for nominative and genitive will occur only if nothing special happens (there is no numeral higher that 5). This will be ensured by three groups of systems:

1. System determining wish of the higher rank (`Thing-Case`)
2. Systems determining if special conditions occur
3. Systems performing real inflection using previous systems as input

First, the system determining wish of the higher rank:

```
Thing-Case (Nominal-Group)
    [Thing-Case-PreNom]
    [Thing-Case-PreGen]
    [Thing-Case-PreDat]
    [Thing-Case-PreAcc]
    [Thing-Case-PreVoc]
    [Thing-Case-PreLoc]
    [Thing-Case-PreIns]
```

The system is similar to the previous version of the system `Thing-Case`, but it has a different names of features and the inflection was removed. The higher rank has to use use preselections with `Thing-Case-PreNom`, etc. instead `Thing-Case-Nom`, etc. Idea behind the names of the features `Thing-Case-PreNom/Gen/`...:

1. It will be nominative/genitive/..., if nothing special happens (Currently, changes can happen only for nominative and accusative).
2. It is not the only way how realize this case (Currently only for genitive).

Now we come to the second group – to the systems determining if genitive should be used instead of nominative or accusative.

To simplify input conditions in the rest of the systems we merge the features `Thing-Case--PreNom` and `Thing-Case-PreAcc` into the feature `NumerativableCases`:

```
NumerativableCases (Thing-Case-PreNom or Thing-Case-PreAcc)
    [NumerativableCases]
```

The following system determines if the `Thing` is numerified:

```
Numerified-Numerativable-Fork (NumerativableCases)
    [Numerified-Numerativable]
    [Not-Numerified-Numerativable]
```

The chooser selecting between two features uses the existing inquiry `Quantification-Q`, which looks into the SPL and determines whether `Thing` has quantity ascription.

```
(CHOOSER
    :NAME Numerified-Numerativable-Fork-Chooser
    :DEFINITION ((ASK (Quantification-Q Thing)
        (NonQuantified (CHOOSE Numerified-Numerativable))
        (Quantified (CHOOSE Not-Numerified-Numerativable))))
)
```

If the object is numerified, the following system will determine if the number is higher than 4:

```
More-Than-Four-Fork (Numerified-Numerativable)
    [More-Than-Four]
    [Not-More-Than-Four]
```

The system uses the following chooser, inquiry and inquiry implementation:

```
(CHOOSER
    :NAME More-Than-Four-Fork-Chooser
    :DEFINITION ((ASK (More-Than-Four-Fork-Q ONUS)
        (More-Than-Four     (CHOOSE More-Than-Four))
        (Not-More-Than-Four (CHOOSE Not-More-Than-Four))))
)
```

```
(ASKOPERATOR
    :NAME More-Than-Four-Fork-Q
    :DOMAIN     KB
    :PARAMETERS (Item)
    :ENGLISH    ( "")
    :OPERATORCODE     KPML::Cz-More-Than-Four-Q-Code
    :PARAMETERASSOCIATIONTYPES      (Concept)
    :ANSWERSET  (Not-More-Than-Four More-Than-Four)
)
```

```
(defun cz-More-Than-Four-Q-Code (item)
    "Is greater than 4?"
    (LET
        ((N (FETCH-SUBC-FEATURE 'Quantity item)))
        (IF (Numberp N)
            (IF (> N 4) 'More-Than-Four 'Not-More-Than-Four)
            'Not-More-Than-Four
        )
    )
)
```

Line `((N (FETCH-SUBC-FEATURE 'QUANTITY ITEM)))` gets the argument of `:quantity` in the SPL and stores it into the variable `N`; if it is greater than 4, `More-Than-Four` is returned (therefore inquiry returns `More-Than-Four`, therefore the chooser chooses feature `More-Than--Four`), otherwise `Not-More-Than-Four` is returned (therefore inquiry returns `Not-More-Than--Four`, therefore the chooser chooses feature `Not-More-Than-Four`).

The following two systems partition numerativable cases into two paths:

1. `Numerative` – numerative case (genitive) will be used
2. `Not-Numerative` – proposed case will be normally used

```
Numerative (NumerativableCases and More-Than-Four)
    [Numerative]



Not-Numerative (NumerativableCases and
        (Not-Numerified-Numerativable or Not-More-Than-Four))
    [Not-Numerative]
```

Finally we have to implement the inflection. We have all the information we need and thus we can simply inflectify depending on input conditions:

```
Thing-Case-Nom (Thing-Case-PreNOM and Not-Numerative)
    [Thing-Case-Nom] inflectify Thing Case-Nom-Form



Thing-Case-Gen (Thing-Case-PreGen or Numerative)
    [Thing-Case-Gen] inflectify Thing Case-Gen-Form



Thing-Case-Acc (Thing-Case-PreAcc and Not-Numerative)
    [Thing-Case-Acc] inflectify Thing Case-Acc-Form
```

For unproblematic cases the the systems are simple:

```
Thing-Case-⟨C⟩ (Thing-Case-Pre⟨C⟩)
    [Thing-Case-⟨C⟩] inflectify Thing Case-⟨C⟩-Form
    where ⟨C⟩ ∈ {Dat, Voc, Loc, Ins}
```

With having all this systems implemented, the grammar for the sentence plan depicted in Figure 19 yields the structure in Figure 20.

```
(EXAMPLE
    :NAME   Quant-Q-3
    :set-name Quant
    :generatedform   "Určete sedm bodů."
    :TARGETFORM      "Určete sedm bodů."
    :GLOSS
        (:ENGLISH "Specify 7 points."
         :LIT     "Specify-P2 7-XX4 points-FP2.")
    :LOGICALFORM
        (S / DM::SPECIFY
            :SPEECHACT IMPERATIVE
            :ACTEE (P / DM::POINT  :QUANTITY 7 )
        )
)
```

Figure 19: SPL for the sentence in Figure 20

7 Evaluation

During the project we performed a rigid evaluation of the system in two steps:

- Evaluation of intermediate prototype in 1999 (see [18])
- Evaluation of the final prototype in September 2000 (see [5])

The evaluation of the intermediate prototype served especially as a feedback for us, to find the weakest points of the system to be able to focus on them in the second phase of the project. In the following paragraphs we focus mainly on the evaluation of the final prototype – it was more elaborate and rigid.

Evaluation addressed the following points:

- System usability – by IT specialists (programmers, students, professors)
- Text quality
  - Acceptability – by professional translators/writers of technical texts
  - Grammaticality – by linguists
- Linguistic coverage

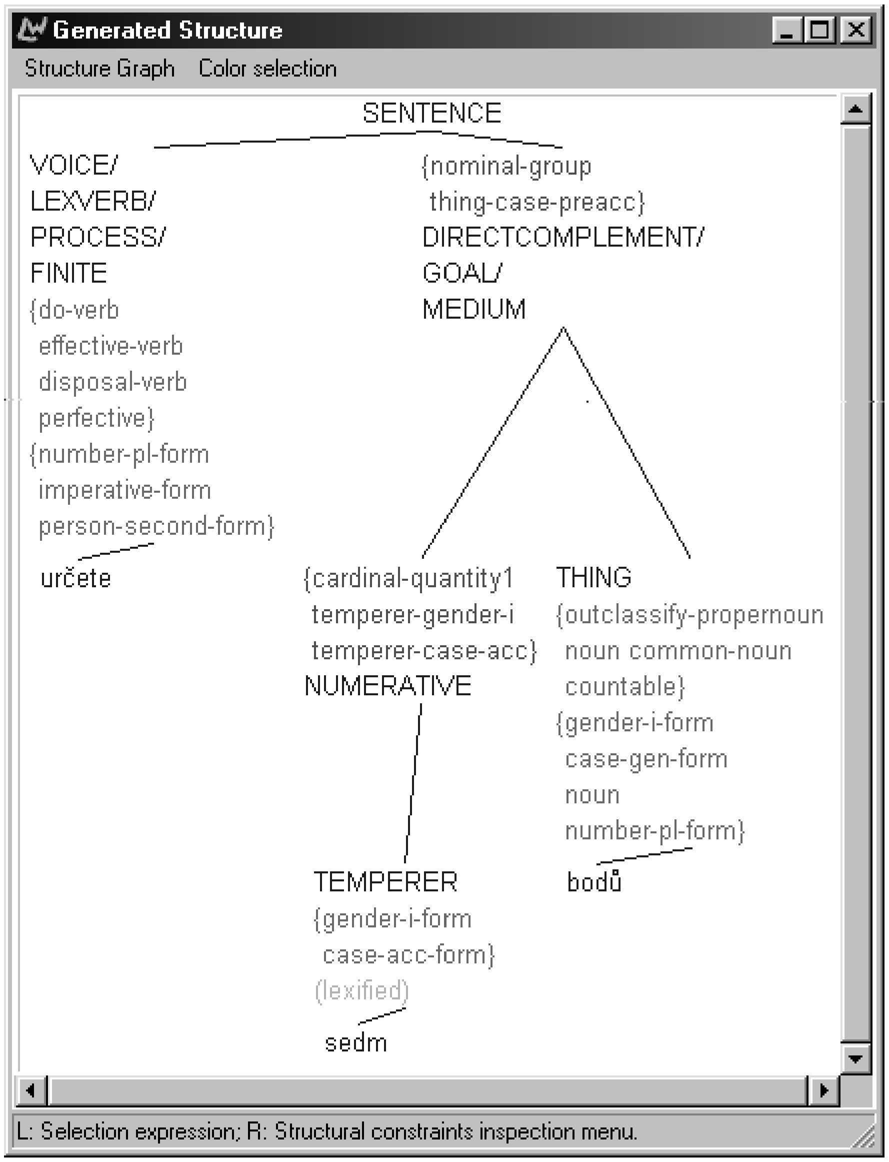
We now describe each part of the evaluation in more detail.

7.1 Usability of the system

Usability evaluation was performed in parallel at all sites. The evaluators were IT specialists (six at each site), some of them having experience from intermediate prototype evaluation.

The evaluation consisted of two main stages:

1. Training of AGILE concepts and in the use of the system. Conceptual Tutorial and Training Manual (describing creation of a sample content model step by step) were used.



SENTENCE

VOICE/
LEXVERB/
PROCESS/
FINITE
{do-verb
effective-verb
disposal-verb
perfective}
{number-pl-form
imperative-form
person-second-form}
určete

{nominal-group
thing-case-preacc}
DIRECTCOMPLEMENT/
GOAL/
MEDIUM

{cardinal-quantity1
temperer-gender-i
temperer-case-acc}
NUMERATIVE

TEMPERER
{gender-i-form
case-acc-form}
(lexified)
sedm

THING
{outclassify-propernoun
noun common-noun
countable}
{gender-i-form
case-gen-form
noun
number-pl-form}
bodů

L: Selection expression; R: Structural constraints inspection menu.

Určete	*sedm*	*bodů.*
Specify	seven-Acc	points-Gen.

‘Specify seven points.’

Figure 20: Accusative with numeral above 4

2. Evaluation proper – each evaluator received five tasks – to edit or create simple or complex models; each task had its time limit. Each evaluator used the system in his/her native language. To test multilinguality, one model was started at one site, than sent to another site, and finished there.

The resulting models were compared to the ideal model and all errors were precisely classified and statistically processed. In general, the structure of the models was correct. Most of the errors can be classified as "typos". However, in some cases evaluators wrongly created multiple instances of a concept instead of multiple pointers to a single instance (which is important for topic-focus articulation and anaphoric reference).

The system interface was judged by evaluators as sufficient to create all desired models, but slow, non-standard and lacking many (especially time-saving) functions common in other development environments. Evaluators especially pointed out that intuitively small changes in existing models are too complicated to realize.

7.2 Text quality – acceptability

The group of evaluators consisted of native speakers experienced in writing or translating software manuals (six persons at each side). They were told that the texts were produced by a (human) translator. A four-point rating scale was used: Excellent, Good, Poor, Terrible. Evaluators also compared the Full Instructions with texts from real manuals (without knowing which one is which).

Functional Descriptions, Full Instructions and Quick References were rated Excellent to Good, while Overviews, were rated Good to Poor. The very encouraging message for the AGILE team was that often (especially in Czech) the generated Full Instructions text was rated better than the original manuals. In Czech, quality was rated as the same in 35% of cases, the manual was better in 24% and the generated text in 41%.

7.3 Text quality – grammaticality

For each of the three languages, we obtained judgments from two linguists (native speakers). They received all of the running-text types (Overviews, Full Instructions, Quick Reference texts and Functional Descriptions). For Bulgarian and Russian, the text types were available in two styles: Personal and Impersonal. For Czech, three styles were available: Personal Indicative (stiskneme), Personal Explicit (stiskněte) and Impersonal Explicit (stiskne se). For all three languages and styles, the texts were generated from the same Model Set.

The evaluators did not find any grammatical errors, but word order issues. However, even these errors were considered to be of stylistic rather than syntactic nature.

7.4 Linguistic coverage

During the development of the system, we employed a method that was both instance-oriented and system-oriented. The primary goal was coverage of the sub-language of CAD-CAM manuals. However, we tried to be as general as possible. The system covers all the target text and large part of instructional passages in (software) manuals (Of course disregarding the lexicon and concepts). The most restricting component of the system are the configurational concepts of the T-box. The grammars are able to generate a much larger set of expressions than the Agile A-box is able to specify.

8 Acknowledgement

I would like to thank the whole AGILE team, especially Hana Skoumalová, Ivana Kruijff-Korbayová, Geert-Jan Kruijff, and John Bateman.

References

[1] John A. Bateman, *KPML Development Environment*, GMD-Studien 304, GMD Forschungszentrum, Informationstechnik GmBH, , 1996.

[2] John A. Bateman, R. Kasper, J. Moore, and R. Whitney, *A general organization of knowledge for natural language processing: the Penman Upper Model*, Tech. report, USC/ISI, , 1990.

[3] Jan Hajič and Barbora Hladká, *Probabilistic and rule-based tagger of an inflective language — a comparison*, Proceedings of ANLP'97, 1997, pp. 111–118.

[4] Tony Hartley, Ivana Kruijff-Korbayová, Geert-Jan Kruijff Danail Dochev, Ivan Hadjiiliev, and Lena Sokolova, *Text structuring specification for the final prototype*, Tech. report, ITRI, University of Brighton, United Kingdom, January 2000.

[5] Tony Hartley, Donia Scott, Ivana Kruijff-Korbayová, Serge Sharoff, Lena Sokolova, Danail Dochev, Kamenka Staykova, Martin Čmejrek, Jiří Hana, and Elke Teich, *Evaluation of the final prototype*, Tech. report, ITRI, University of Brighton, United Kingdom, October 2000.

[6] Petr Karlík, Marek Nekula, and Zdenka Rusínová, *Příruční mluvnice češtiny*, Nakladatelství Lidové Noviny, Praha, 1997.

[7] Geert-Jan Kruijff and Ivana Kruijff-Korbayová, *Text structuring in a multilingual system for generation of instructions*, In Matoušek et al. [15], pp. 89–94.

[8] Geert-Jan Kruijff, Ivana Kruijff-Korbayová, and John Bateman, *The text structuring module for the intermediate prototype*, Tech. report, ITRI, University of Brighton, United Kingdom, July 1999.

[9] Geert-Jan M. Kruijff, Ivana Kruijff-Korbayová, Serge Sharoff, Ivan Hadjiiliev, Lena Sokolova, and Michael Boldasov, *Flexible text structuring for the final prototype*, Tech. report, ITRI, University of Brighton, United Kingdom, June 2000.

[10] Ivana Kruijff-Korbayová, John Bateman, and Geert-Jan M. Kruijff, *Generation of contextually appropriate word order*, Information Sharing (Kees van Deemter and Rodger Kibble, eds.), Lecture Notes, CSLI, in prep.

[11] Ivana Kruijff-Korbayová and Geert-Jan Kruijff, *Handling word order in a multilingual system for generation of instructions*, In Matoušek et al. [15], pp. 83–88.

[12] Ivana Kruijff-Korbayová, Geert-Jan Kruijff, John Bateman, Danail Dochev, Nevena Gromova, Tony Hartley, Elke Teich, Serge Sharoff, Lena Sokolova, and Kamenka Staykova, *Specification of elaborated text structures*, Tech. report, ITRI, University of Brighton, United Kingdom, April 1999.

[13] W. C. Mann and C. M. I. M. Mathiesen, *Demonstration of the Nigel text generation computer program*, Systemic Perspectives on Discourse (J. D. Benson and W. S. Greaves, eds.), vol. 1, 1985, pp. 50–83.

[14] William C. Mann and Susan A. Thompson, *Rhetorical structure theory: Toward a functional theory of text organization*, Text **8** (1987), no. 3, 243–281.

[15] Václav Matoušek, Pavel Mautner, Jana Ocelíková, and Petr Sojka (eds.), *Proceedings of the Conference on Text, Speech and Dialogue (TSD'99), Mariánské Lázně, Czech Republic*, Springer-Verlag, 1999.

[16] Richard Power, *Preliminary model of the CAD/CAM domain.*, Tech. report, ITRI, University of Brighton, United Kingdom, June 1998.

[17] ______, *Final model of the CAD/CAM domain.*, Tech. report, ITRI, University of Brighton, United Kingdom, July 1999.

[18] Serge Sharoff, Donia Scott, Tony Hartley, Danail Dochev, Jiří Hana, Martin Čmejrek, Geert-Jan Kruijff, and Ivana Kruijff-Korbayová, *Evaluation of the intermediate prototype*, Tech. report, ITRI, University of Brighton, United Kingdom, April 2000.

[19] Serge Sharoff, Lena Sokolova, Danail Dochev, Ivana Kruijff-Korbayová, Jiří Hana, Geert-Jan Kruijff, Kamenka Staykova, Elke Teich, and John Bateman, *Formal specification of full grammar models and implementation of tactical generation resources for all three languages in a final prototype*, Tech. report, ITRI, University of Brighton, United Kingdom, July 2000.