A Student Dictionary System

Fabrizio Caruso Carsten Schneider Geert Van de Weyer
under the guidance of Wolfgang Schreiner

April 21, 1999

Abstract

In this paper we present a detailed case study of formal systems specification with the aid of the term rewriting system cafeOBJ. The software system we use for this case study is a rather straightforward and uncomplicated system that we call a *Student Dictionary System*. This is basically an electronic dictionary that has some functions allowing its users to test their knowledge of the vocabulary of a certain language.

We demonstrate, using this example, how an informal notion of the functionality of a software system is translated into a formal formulation of the functionality of this software system using notions from the theory of formal systems specification. Furthermore, we demonstrate how implementing this formal specification in a term rewriting language helps the software designer to test the current functionality of his or her system and in that way helps him or her to resolve ambiguities in the initial design of the system.

Contents

Chapter 1

Introduction

In this paper we will present a (detailed) case study of the use of algebraic specification of abstract data types; we will use the loose specification of an abstract data type of what we chose to call a *Student Dictionary System* and an implementation of this specification in the cafeOBJ term rewriting system [NSF98]. A loose specification [LEW96] is a more or less abstract form of an algebraic specification, i.e. the formulation of the data type is already rather unambiguous, but is not yet formulated in such a way that the system presented in this way is executable. This paper is the final result of a student project in the scope of the course "Formal Specification of Abstract Data Types" held at RISC-Linz by Wolfgang Schreiner in the winter semester of 1998. Some additional information can also be found on the URL `http://www.risc.uni-linz.ac.at/courses/ws98/specadt/`.

Before proceeding any further, let us briefly introduce the notion of an algebraic specification of a data type and give a hint to its importance in the design of software. Algebraic specifications appear at a rather crucial point in the so-called *software life cycle*. More precisely, they are a tool to translate an informal notion of the functionality of a software system into a formal one before actually starting to encode this specification. Thus it allows the software designers to check whether their current idea of a program will actually provide the functionality wanted of the program (so-called rapid prototyping). Furthermore, the rather unambiguous way of formulating the design of their program allows the software designers to resolve ambiguities and flaws at a rather early stage. While the *loose* specification is already *precise*, a corresponding *initial* specification may be even *executable*; we have implemented such a specification in the cafeOBJ term rewriting system [NSF98]. For further information on the theory of algebraic specification, we refer the reader to [LEW96]. For a comparison with other formal specification techniques, see [PST96] and [GM85].

The *Student Dictionary System* presented here is based on the Word-for-Word system presented in Chapter 3 of [PST96]. Our intention is to use the case study presented in this paper to shed some light on the different aspects involved when designing a formal specification of a data type. In particular, our presentation of this algebraic specification will clarify (to some extend) the interplay between designing a specification without using any software tools and testing the (partially) finished specification with term rewriting tools such as the cafeOBJ language. Furthermore, we intend to use the implementation of our specification in the cafeOBJ language to illustrate some of the difficulties that arise when one wants design and implement a formal specification of an abstract data type completely.

The implementation of this specification in the cafeOBJ language not only allows us to clarify the interplay between designing an algebraic specification and testing a rough implementation of this specification, but also gives us a (rough) tool that enables us to provide examples illustrating the functionality of the system we specified. The code for this cafeOBJ implementation is available on the URL `http://www.risc.uni-linz.ac.at/people/fcaruso/Spec.html`.

We have structured this paper in the following way: The second chapter contains the entire loose specification, cut into parts. For each of these parts, we give the main ideas behind this specification. The third chapter then contain the cafeOBJ implementation of the specification and demonstrates the working of the dictionary by some examples. The fourth and last chapter then contains the main conclusions we have drawn.

Chapter 2

The Mechanics of the Specification

Let us now sketch our interpretation of a *Student Dictionary System*, before we will detail the formal specification of this system in the rest of the paper. The primary purpose of the system we will present is to allow its user to test his current knowledge of a certain foreign language by generating a test suite. Such a test suite provides a number of words to the user and expects the translations of these words in return. Using the answers of the user, it then computes a relative score, representing the user's knowledge of the vocabulary of the foreign language.

Before the user can test his knowledge of the vocabulary of a certain language, he has to be able to provide the computer with the vocabulary list that will be used to test his knowledge. For this reason the system we will specify also has operations that allow the user to construct vocabulary lists. In addition to this, the system also provides the basic operations everybody expects from a dictionary: providing translations of a given word or providing alphabetically ordered vocabulary lists.

2.1 Drawing the Plans: Analyzing the Informal Description

The very first step in constructing a formal specification of an abstract data type is the analysis of the informal description, in conjunction with the (rough) design of the layout of the formal description (or specification). The specification as sketched in the following paragraphs more or less represents the process we followed in (roughly) formalizing the initial informal description we received. We first of all tried to work out what the very core of the system we were to design would be, and came up with the *vocsystem*. Using this core as a starting point, we set out to formalize the functionality of the *Student Dictionary System*, working in two directions: one direction leading to what would finally become the user interface, and the other direction leading to the very basics that comprise the core of the system (the languages and words in languages, lists and pairs, etcetera).

Let us first of all explain the main idea behind the entire specification. The very core of the dictionary system is the sort *vocsystem* (short for vocabulary system). This sort has two major components: a couple of languages, and a list of vocabulary items. Of this couple of languages, the first is called the native language and the second is called the foreign language. The list of vocabulary items then contains the actual data of the dictionary, which is represented by the sort *vocitem* (short for vocabulary item).

This sort *vocitem* actually has two components. The first component is a pair of words, its first entry containing a native word and its second entry containing the foreign translation of the first entry. The second component of a *vocitem* is a boolean value. This value is used when generating a test suite: it tells the system whether the vocabulary item is already in the test suite or not.

It is clear from this description that the sorts *vocsystem* and *vocitem* are composed out of other sorts, representing the brickwork that is always used to build any dictionary, electronically or not. This brickwork, of course, are the languages and the words in which the languages are written. In our loose specification, we introduced the sorts *language* and *word* to represent these basic concepts of any dictionary.

We chose to work with this sort *word* internally instead of working with the simpler sort *string* for a very natural reason: the notion of *word* which we introduced has apart from a string also a component *language*. This allows us to check the spelling of a string when making it into a word, avoiding tedious spelling checks throughout the dictionary later on.

The *language* sort then contains the two final aspects one needs to build up a dictionary. These are the characters and their ordering, and the lexicographical rules, telling which character combinations cannot occur in a language. The purpose of these lexicographical rules is to avoid often occuring typing mistakes.

The build-up explained in the previous paragraphs is visualized nicely in Figure 2.1.

Apart from the vocabulary system *vocsystem* there are two other important sorts for the user. These are the sorts *testnativevocsystem* and *testforeignvocsystem* that represent the test suite a user can construct to test his knowledge of a given language. These two sorts basically consist out of a list of words (either native or foreign) and one integer. This integer represents the score of the user when he goes through the test.

This concludes the basic idea which lies behind the loose specification. Let us now have a look at the detailed operations of the specification. Doing this, we will adopt a bottom-up approach to the problem, although the actual design of the specification happened more or less top down. Working bottom-up, however, allows us to build up the dictionary from scratch.

2.2 Tools for the Weary: Lists and Boolean

When we started the design of the Student Dictionary System, we more or less immediately decided that any dictionary is nothing more than a complicated

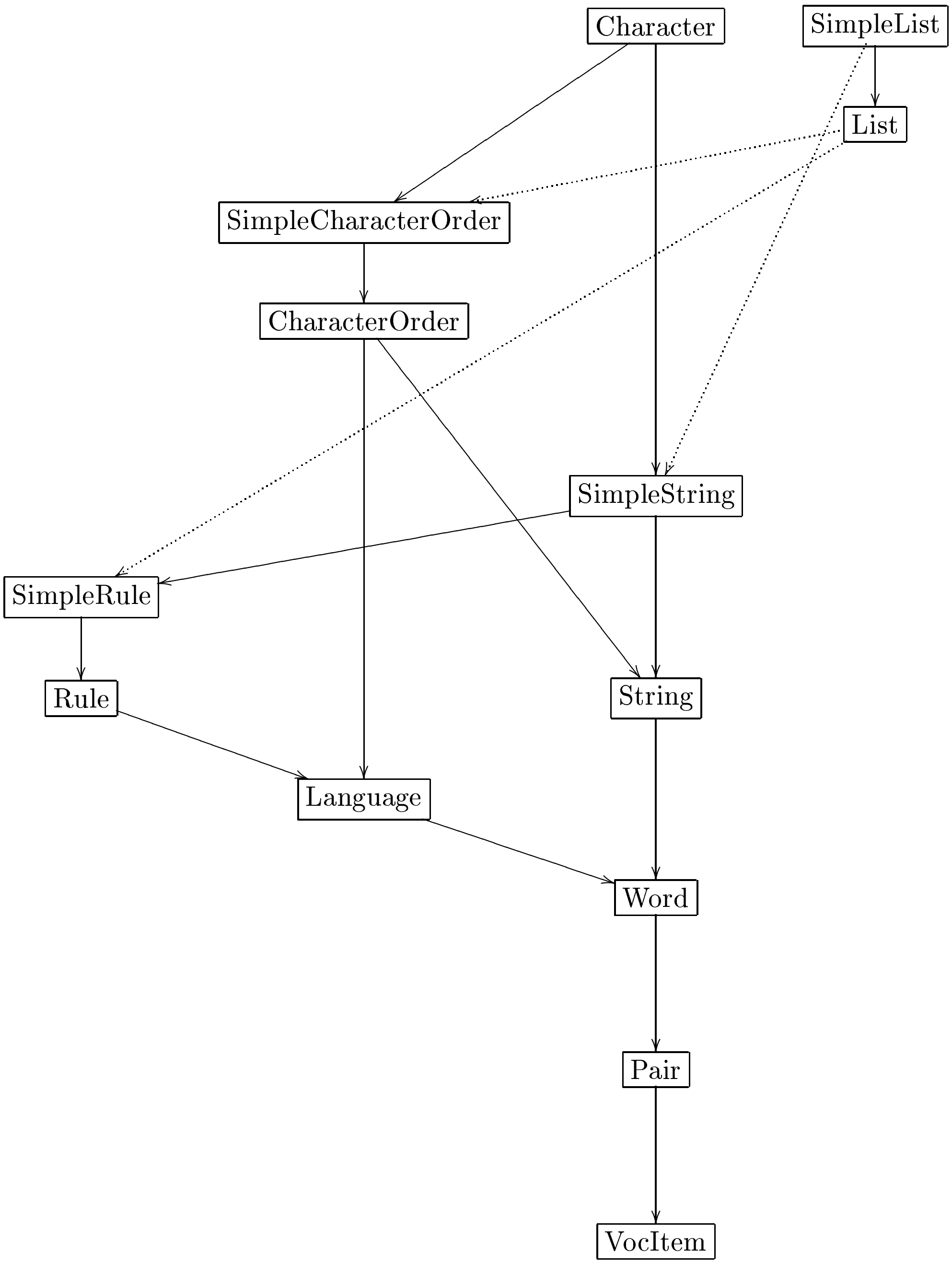

Figure 2.1: The structure of the lower level of the Student Dictionary System

type of list. With this in our mind, we decided that we should create the dictionary system by extending a basic list sort. This approach turned out to be useful, since we also could use this list to specify our notion of an alphabet, as is seen later on in this paper. But let us now specify our basic list sort.

First of all, we need the notion of an element. An element can be any arbitrary object at this point, so its loose specification only contains a type, and no operations or axioms. This sort only serves as a placeholder and will be replaced by more explicit sorts when we will use the list type constructed in this section in order to create more elaborate sorts needed at a higher level in the student dictionary system. The loose specification for the sort element becomes:

spec ElementSpec

type element

endspec

The fact that the sort element we just introduced is a placeholder becomes clearer when we take a look at the first list sort we create, in the specification *SimpleListSpec*. This specification contains the standard list as we all know it from computer science, extended with operations that allow us to get the first element of the list or remove the first element from the list (Head resp. Tail), concatenate lists (_._), check whether an element belongs to a list or not ($\in$) and tell us the length of a list (Length). For these last two operations, we need to import the boolean values and the integers into this specification, as can be seen in the import section of the specification *SimpleListSpec* given below.

spec SimpleListSpec(E: ElementSpec)

import E, BoolSpec, IntegerSpec

freely generated type list

opns

constr [ ]: $\rightarrow$ list

constr AddToSimpleList: element$\times$list$\rightarrow$ list

[_]: element$\rightarrow$ list

Head: list$\rightarrow$ element

Tail: list$\rightarrow$ list

_._: list$\times$list$\rightarrow$ list

_$\in$_: element$\times$list$\rightarrow$ boolean

Length: list$\rightarrow$ int

vars

l,m,n: list

e:element

axioms

Head(AddToSimpleList(e,l))=e

Tail(AddToSimleList(e,l))=l

[e]=AddToSimpleList(e,[ ])

[ ].l=l

AddToSimpleList(e,l).m=AddToSimpleList(e,l.m)
e∈l=**If** ∃ m,n: (l=m.[e].n)
    then True **else** False
Length([ ])=0
Length(AddToSimpleList(e,l))=Succ(Length(l))

endspec

This first specification of list will now be extended to a list that does not contain the same element twice. In order to create this list sort out of the previous sort we only have to provide one additional operation, taking the role of AddToSimpleList in the previous specification. This operation, named AddToList, only adds an element to a list if this element was not in the list before, as is seen in the axioms of the specification *ListSpec* given below. We realized we were going to need this property when we decided to represent an alphabet using lists: this meant that we could not allow multiple occurences of the same character. It also became clear that the input of the user, answering questions in the test suite should be of this nature. This avoided modifying the score multiple times when the same answer was entered multiple times.

spec ListSpec(E: ElementSpec)
  extends SimpleListSpec(E) by

  opns
    AddToList: element×list→ list

  vars
    e:element
    l:list

  axioms
    AddToList(e,l)=
      If $e \in l$ **then** l **else** AddToSimpleList(e,l)

endspec

It is now clear from the previous specifications that the boolean values play an important role in this specification, as they allow us to check (among other things) whether elements are in a list or not, so we provide them in the next specification:

spec BooleanSpec
  freely generated type boolean

  opns
    constr True: → boolean
    constr False: → boolean

endspec

2.3 Clay for the Bricks: Characters and Character Ordering

As we explained in the previous section, our specifications of the sorts *list* and *boolean* will be used as tools to build up the student dictionary system. In this section we will put them to use for the first time, building the notion of a character ordering. This is but the first step in our construction of a student dictionary system, and is still concerned with the very basics of such a system.

The very basic components of our (written) language are of course the characters making up the language. We introduce the characters here in the specification *CharacterSpec* as a freely generated sort containing only constants. Our intention here was to provide the user not only with the standard Roman alphabet, but also allow other characters like Greek, Cyrillic or Chinese.

> **spec** CharacterSpec
>
> freely generated type char
>
> **opns**
>
> constr a: $\rightarrow$ char
>
> constr b: $\rightarrow$ char
>
> $\vdots$
>
> **endspec**

Closely linked to the characters is of course the alphabet, which simply lists all possible characters in a certain order. This order is then later on used to order words made up of these characters. This ordered list of characters is the sort we call *charorder* and is simply a list of characters where we renamed some sorts and operation and added the operation *IsCharSmaller* representing the order of the characters. So it is really an alphabet in the sense that it contains all characters and has an ordering on these characters.

The modification of the sort *list* to the sort *charorder* is done in the specification *CharacterOrderSimpleSpec*:

> **spec** CharacterOrderSimpleSpec
>
> import ListSpec(CharacterSpec {element $\rightarrow$ char}) by renaming
>
> list to charorder
>
> - AddToList to AddToOrder
> - AddToSimpleList to NewChar
>
> **endspec**

The specification *CharacterOrderSimpleSpec* is then extended with an ordering, represented by the operation *IsCharSmaller*, in the specification *CharacterOrderSpec*. This operation simply tell us that the characters appear in ascending order in the list containing all characters. Note that we can do this since the list sort we use cannot contain the same element twice.

> **spec** CharacterOrderSpec

import BoolSpec

extends CharacterOrderSimpleSpec by

opns

IsCharSmaller: char$\times$char$\times$charorder$\rightarrow$ boolean

vars

c1,c2: char

co,co1,co2,co3: charorder

axioms

IsCharSmaller(c1,c2,co)=

If $\exists$ co1,co2,co3: (co1.[c1].co2.[c2].co3=co)

then True **else** False

endspec

The notion of smaller we introduced here should more accurately be called strictly smaller, since this operation returns false when we check whether a certain character is smaller than itself. Also note that we do not take into account the fact that maybe some characters are not in the alphabet at all: if this is the case, the operation specified here will simply return the boolean value False.

2.4 Baking the Bricks: Strings and Rules

In the previous section, we constructed the raw material for the design of a language, which is the next step in the construction of a dictionary. We will now use this raw material, together with the earlier developed tool *list* to design more advanced material for our construction. This more advanced material are the strings and lexicographic rules. We think the notion of a string is pretty straightforward: it is simply a finite sequence of characters, and will of course be used to represent words from a language later on. The concept of lexicographic rules, however, is not that straigthforward, and we chose to interpret this in the following way: lexicographic rules represent sequences of characters that cannot occur in a given language.

For specifying strings, we make use of the earlier introduced sorts *list* and *charorder*. We begin by stating that basically a string is nothing else but a simple list of characters (where simple list means that we allow multiple occurences of the same character). This statement is formalized in the specification *SimpleStringSpec*.

spec SimpleStringSpec

import SimpleListSpec(CharacterSpec {element $\rightarrow$ char}) by renaming

list to string

- [ ]to ϵ
- AddToSimpleList to AddToString

endspec

However, instead of simply stating that a string is a list of characters, we provide additional operations that represent the basic notion of alphabetical ordering. For this purpose we provide the operation *IsStringSmaller*. We also provide an additional operation that will be used later on in the specification to change a string into a correctly spelled word. This operation, *CheckAlphabet*, checks whether a given string is written in a correct alphabet. There is also an error operation which is used later on in the specification (more precisely in the specification *WordSpec*). This extension is done in the specification *StringSpec*.

```
spec  StringSpec
    imports CharacterOrderSpec
    extends SimpleString by
    opns
        Error_NoStringLeft: → string
        IsStringSmaller: string×string×charorder→ boolean
        CheckAlphabet: string×charorder→ boolean
    vars
        s1,s2:string
        c:char
        co:charorder
    axioms
        IsStringSmaller(s1,s2,co)=
                If s1=ϵ then True
                  else
                    If s2=ϵ then False
                      else
                        If Head(s1)=Head(s2)
                          then IsStringSmaller(Tail(s1),Tail(s2),co)
                          else IsCharacterSmaller(Head(s1),Head(s2))

        CheckAlphabet(ϵ,co)=False
        CheckAlphabet(AddToString(c,ϵ))=
                If c ∈ co then True else False
        CheckAlphabet(AddToString(c,s1),co)=
                If c ∈ co
                  then CheckAlphabet(s1,co) else False
endspec
```

Note that at this point we do check whether all characters occuring in a string also occur in a given alphabet. This error checking takes care of the fact that our operation *IsCharSmaller* does not consider comparing characters from two different alphabets a problem: the operation *IsCharSmaller* will only be used for ordering words, and at this point a check for alphabet correctness has already been done.

In the specification of the ordering *IsStringSmaller* we made a mistake: instead of specifying a strict ordering ($<$), we specified the ordering $\leq$. This mistake did not manifest itself when we implemented the specification in cafeOBJ, because we used the predefined character ordering provided by cafeOBJ in the implementation.

As stated earlier, we also introduce lexicographic rules. Once again, we do this through the extension of the sort list. Basically, the lexicographic rules are a list of strings, as is seen in the specification *SimpleRuleSpec*. Each string occuring in this list represents a sequence of characters that **cannot** occur in the language to which the lexicographic rules belong.

spec SimpleRuleSpec

import ListSpec(StringSpec {element → string}) by renaming

- list to rules
- AddToList to AddToRules
- AddToSimpleList to NewRule

endspec

We provide an operation *MatchRule* to check whether a given string matches a lexicographic rule or not. If this is the case, the operation returns True, which means that the given string automatically represents an **incorrect** word for the language to which the rules belong.

spec RuleSpec

extends SimpleRuleSpec by

opns

MatchRule: string×rule→ boolean

vars

s,s1,s2:string

R:rules

axioms

MatchRule(s,R)=

If R=[ ]**then** False

else

If ∃ s1,s2: (s1.[Head(R)].s2=s) **then** True

else MatchRule(s,Tail(R))

endspec

Note that the axiom does not take the fact into account that the string checked could be written in an alphabet which is completely different from the alphabet of the language to which the lexicographic rules belong. We did not provide a check for this phenomenon because strings provided by the user to the system represent strings in a specific language. This language contains both an alphabet and lexicographic rules. Any string the user enters representing a word in this language will be checked for correctness in the use of characters by *CheckAlphabet* and *MatchRules*.

2.5 Bricks to Build With: Word and Language

Now that we have an alphabet and rules, we basically have all the ingredients for a language. The notion of language that we will use is simply a couple of an alphabet and rules. This is seen in the specification *LanguageSpec* (containing also operations to extract the lexicographic rules and the characters out of a given language).

spec LanguageSpec

import RuleSpec, CharacterOrderSpec

freely generated type language

opns

constr MakeLanguage: rules×charorder→ language

GetRules: language→ rules

GetCharacterOrder: language→ charorder

vars

co:charorder

r:rules

axioms

GetRules(MakeLanguage(r,co))=r

GetCharacterOrder(MakeLanguage(r,co))=co

endspec

Note, however, that we overlooked a very crucial error that could be made here: we do not check whether the characters in the alphabet and the characters in the lexicographic rules are the same! This means that our entire notion of error checking, which is based on this very concept as we made clear when we discussed the operations *CheckAlphabet* and *MatchRule*, will not work correctly if the alphabets differ. One way to correct this error is using the operation *CheckAlphabet* on all strings occuring in the rules specification and returning an error if one of these strings does not match the alphabet given as a character ordering to the constructor *MakeLanguage*.

This language is now used to make correctly spelled words. Up until now, we only had sequences of characters and the possibility to order them. In our dictionary, however, we want to work with correctly spelled words, so we combine the means we have for spelling checking with the means we have for making strings. This results in the sort *word*. This sort is basically a couple consisting out of a string and a language and checks whether the given string is correctly spelled in the given language, using the operations *MatchRules* and *CheckAlphabet* (modulo the error we mentioned in the previous paragraph).

spec WordSpec

freely generated type word

import StringSpec, LanguageSpec

opns

```
        MakeWord: string×language→ word
        constr CorrectWord: string→ word
        constr Error_Misspelled: → word
        constr Error_NoWordLeft: → word
        constr DummyValue: → word
        GetString: word→ string
    vars
        s:string
        l:language
    axioms
        GetString(CorrectWord(s))=s
        GetString(Error_NoWordLeft)=Error_NoWordLeft
        MakeWord(s,l)=
            If MatchRule(s,GetRules(l))=False &&
              CheckAlphabet(s,GetCharacterOrder(l))=True
                then CorrectWord(s) else Error_Misspelled
endspec
```

GetString(Error_Misspelled) is intentionally left unspecified. Two constructors, *Error_NoWordLeft* and *DummyValue* do not appear to have any functionality here. This is true, because they were added during the design of the specification of what we call a testvocabulary system and only have a meaning on this level. This specification is given in Section 2.8.

2.6 Laying the Foundation: Pair, PairList and ListOfPairList

In the previous sections we designed all the basic material one needs to construct a dictionary. In the next sections, we will use this basic material to develop more advanced structures needed to make a student dictionary system.

A dictionary consists out of pairs of words, representing a word in a given language and its translation into another given language. For this purpose, we have to introduce the concept of pairs and lists of pairs, as is seen below.

```
spec PairSpec
    freely generated type pair
    import WordSpec, StringSpec
    opns
        constr MakePair: word×word→ pair
        GetNative: pair→ string
        GetForeign: pair→ string
    hidden opns
        CheckWords: word×word→ boolean
    vars
        n,f: word
        p: pair
    axioms
        CheckWords(n,f)=
          If n=!=Error_Misspelled && f=!=Error_Misspelled
            then True else False

        GetNative(MakePair(n,f))=
          If CheckWords(n,f)=True then GetString(n)
        GetForeign(MakePair(n,f))=
          If CheckWords(n,f)=True then GetString(f)
endspec
```

Note that in the axioms for the operations *GetNative* and *GetForeign* we used if-then statements. What we actually mean by this is the following:

$$\mathsf{GetNative(MakePair(n,f))} = \mathsf{GetString(n)}\ \mathbf{if}\ \mathsf{CheckWords(n,f)} = \mathsf{True}$$

and the same statement for *GetForeign*. This means that *GetNative* and *GetForeign* are unspecified when the given words in the pair are mispelled.

It is clear from this specification that a pair is simply a couple of words, the first element of this couple is called the native word whereas the second element is called the foreign word. We also provide an operation to check whether the pair contains the error message introduced in the word specification.

```
spec PairListSpec
    import ListSpec(PairSpec {element → pair}) by renaming
      • list to pairlist
      • AddToList to AddPair
endspec
```

Once again, we use the specification for a list, introduced earlier on in this paper, to create list of pairs, as is seen in the specification *PairListSpec*.

Of course one word can have many different translations, and it is useful to group all these translations. If the user then wants to see the entire dictionary, he is returned a list of these groups. For this purpose we introduce a list of a list of pairs, again using the list specification introduced earlier.

spec ListOfPairListsSpec

import ListSpec(PairListSpec {element → pairlist}) by renaming

- list to listofpairlists
- AddToList to AddPairList
- AddToSimpleList to NewPair

endspec

2.7 A Skeleton Arises: VocItem and ListOfVocItem

Our ultimate goal is to construct test suites out of a given dictionary. This means we have to select several entries out of the dictionary, which we use to test the user. Of course we do not want to ask the user the same question several times, so we have to indicate whether a certain entry has been chosen or not. For this purpose, our dictionary does not simply contain pairs of words, but contains so-called vocabulary items. These *vocitems* contain in addition to a pair also a boolean flag, which is set to true if the pair is in the testsuite that is generated. These vocitems are introduced in the *VocItemSpec* specification below.

spec VocItemSpec

freely generated type vocitem

import PairSpec

opns

constr MakeItem: pair×boolean→ vocitem

GetPair: vocitem→ pair

GetFlag: vocitem→ boolean

vars

vi:vocitem

p:pair

f:boolean

axioms

GetFlag(MakeItem(p,f))=f

GetPair(MakeItem(p,f))=p

endspec

Of course our dictionary will then contain a list of these vocabulary items, specified in the *ListOfItemsSpec* specification below.

spec ListOfItemsSpec

import ListSpec(VocItemSpec {element → vocitem}) by renaming

- list to listofitems
- AddToList to AddItem
- AddToSimpleList to NewItem

endspec

For the user-interface we will also need data types which are specified in the following specifications. The *ListOfStringsSpec* provides the user with a tool to enter his translations of a given word in a test suite. Multiple entries of the same translation will be avoided; in this way we aviod checking whether a translation has already been provided by the user when counting his score.

spec ListOfStringsSpec

import ListSpec(StringSpec {element → String}) by renaming

- list to listofstrings
- AddToList to AddString
- AddToSimpleList to NewString

endspec

spec ListOfStringListSpec

import ListSpec(ListOfStringsSpec {element → listofstrings}) by renaming

- list to listofstringlist
- AddToList to AddStringList
- AddToSimpleList to NewStringList

endspec

spec ListOfListOfItemsSpec

import ListSpec(ListOfItmesSpec {element → listofitems}) by renaming

- list to listoflistofitems
- AddToList to AddToItemListsList
- AddToSimpleList to NewStringListsList

endspec

After designing these specifications, we implemented them in the cafeOBJ language and tested the implementation. We did not, however, notice the error in the specification of the sort language until Wolfgang Schreiner pointed this error out to us during the presentation of the student dictionary system. The cafeOBJ implementation containing all the previous specifications is called `lowlevel.mod`.

2.8 A Dictionary Has Been Built: DictionarySpec

We now have introduced all the basics we need to create the core of our student dictionary, namely the vocabulary system. The essence of this sort is that it consists out of a couple of languages and a list of vocabulary items. In addition to this, we provide all operations that are nescessary to build this dictionary from scratch and operations that provide us with information on the current dictionary.

spec DictionarySpec

freely generated type vocsystem

import ListOfItemsSpec

opns

constr Dictionary: language×language×listofitems→ vocsytem

constr Error_NativeWordIsMisspelled: → vocsystem
constr Error_ForeignWordIsMisspelled: → vocsystem
AddEntryToDictionary: string×string×vocsystem→ vocsystem
DeleteEntryFromDictionary: string×string×vocsystem→ vocsystem
GetNativeLanguage: vocsystem→ language
GetForeignLanguage: vocsystem→ language
GetItemList: vocsystem→ listofitems
RandomVocItem: vocsystem→ vocitem
ResetFlags: vocsystem→ vocsystem
ResetFlagsRec: listofitems×listofitems→ listofitems
SetVocItemFlag: vocitem×boolean→ vocitem
SetNativeFlags: word×vocsystem→ vocsystem
SetNativeFlagsRec: word×listofitems×listofitems→ listofitems
SetForeignFlags: word×vocsystem→ vocsystem
SetForeignFlagsRec: word×listofitems×listofitems→ listofitems
TranslateIntoNative: string×vocsystem→ listofitems
TranslateIntoNativeRec: string×listofitems×listofitems→ listofitems
TranslateIntoForeign: string×vocsystem→ listofitems
TranslateIntoForeignRec: string×listofitems×listofitems→ listofitems

vars

l1,l2: language

s1,s2,s: string
il,output: listofitems
vi: vocitem
voc: vocsystem
w_1,w_2,w: word
b: Bool

axioms

GetNativeLanguage(Dictionary(l1,l2,il))=l1

GetForeignLanguage(Dictionary(l1,l2,il))=l2
GetItemList(Dictionary(l1,l2,li))=il
AddEntryToDictionary(s1,s2,voc) =
  Let w_1:= MakeWord(s1,GetNativeLanguage(voc));
  Let w_2:= MakeWord(s2,GetForeignLanguage(voc));
  If w_1=Error_Misspelled
   then Error_NativeWordIsMisspelled
    If w_2=Error_Misspelled
     then Error_ForeignWordIsMisspelled
     else

```
    Dictionary(GetNativeLanguage(voc), GetForeignLanguage(voc),
     AddItem(MakeItem(MakePair(w1,w2),False),GetItemList(voc)))
DeleteEntryFromDictionary(s1,s2,Dictionary(l1,l2,[ ])) =
  Dictionary(l1,l2,[ ])
DeleteEntryFromDictionary(s1,s2,Dictionary(l1,l2,AddItem(vi,il))) =
  If s1 = s1 and s2 = s2 then Dictionary(l1,l2,il))
  else
   AddEntryToDictionary(GetNative(GetPair(vi)), GetForeign(GetPair(vi)),
     DeleteEntryFromDictionary(s1,s2,Dictionary(l1,l2,il))
SetVocItemFlag(vi,b) = MakeItem(GetPair(vi),b)
SetNativeFlags(w,voc) =
  Dictionary(GetNativeLanguage(voc), GetForeignLanguage(voc),
   SetNativeFlagsRec(w,GetItemList(voc),[ ]))
SetForeignFlags is analogeous.
SetNativeFlagRec(w,il,output) =
  If il = [ ]then output
  else
   Let hdvcitm:= Head(il)
   If GetNative(GetPair(hdvcitm)) = GetString(w) then
    SetNativeFlagsRec(w,Tail(il), AddItem(SetVocItemFlag(hdvcitm),true),output)
   else
    SetNativeFlagsRec(w,Tail(il),AddItem(il,output))
SetForeignFlagRec is analogeous.
ResetFlagsRec(il,output) =
  If il = emptyL then
   output
  else
   Let hdvcitm := Head(il)
   ResetFlagsRec(Tail(il), AddItem(SetVocItemFlag(hdvcitm,False),output))
ResetFlags(voc) =
  Dictionary(GetNativeLanguage(voc), GetForeignLanguage(voc),
   ResetFlagsRec(GetItemList(voc),[ ]))
RandomVocItem(voc)=
   If ∃p ∈ GetItemList(voc): (GetFlag(p)=0)
    then
     MakeItem(p,*)
    else Error_OutOfEntries
TranslateIntoNativeRec(s,il,output) =
  If il = [ ] then output
  else
   Let foreignstr := GetForeign(GetPair(Head(il)))
   If foreignstr = s then
    TranslateIntoNativeRec(s, Tail(il),AddItem(Head(il),output))
   else
    TranslateIntoNativeRec(s, Tail(il),output)
TranslateIntoForeignRec is analogeous.
```

TranslateIntoNative(s,voc) =
  TranslateIntoNativeRec(s,GetItemList(voc),[ ])
TranslateIntoForeign is analogeous.

endspec

* is a dummy value and can be either True or False.

In this specification, we provided operations to extract the languages and the list of vocabulary items from a given vocabulary system; these are the operations *GetNativeLanguage*, *GetForeignLanguage* and *GetItemList*. We also provided operations to add entries to a dictionary (*AddEntryToDictionary*), to translate a given string, representing either a native or a foreign word (*TranslateIntoForeign* resp. *TranslateIntoNative*), as well as operations for more internal use whose purpose it is to set or reset the flags of vocabulary items (*SetVocItemFlag* resp. *ResetFlags*). We also provide constants for error checking (*Error_NativeWordIsMisspelled* and *Error_ForeignWordIsMisspelled*).

There is one operation we should especially mention here, and this is the operation *RandomVocItem*. The notion we used here, "take randomly", can be interpreted in two ways: one could either interpret it as "take any such p you like" or one could really implement a notion of randomness here, using distributions. If one uses the first interpretation, however, the risk of always picking the same collection of words with this operation becomes greater, therefore, when we designed this specification, we had the second meaning in the back of our heads, because we want our system to generate an as diversified collection of tests as possible (and the operation *RandomVocItem* will be used for the construction of these tests).

At this point, we decided to join implementing our student dictionary system in the cafeOBJ language and writing the loose specification together in order to speed up the development of the system. This meant that we started to write the specification in such a way that the cafeOBJ language understood it more or less immediately. This had, however, some drawbacks: cafeOBJ does not allow **if-then-else** statements, so we had to split these up and specify the different possibilities. Another drawback was the fact that cafeOBJ does not allow constants, so we could not assign long operation calls to a short name. This is reflected in the loose specifications in the following part, and not always helps to clarify the specification.

2.9 Framework: NativeVocSystem and TestNativeVocSystem

The basic dictionary that we have built is now used to generate a test suite so that the user can test his knowledge of the language. The construction of this test suite is done in two steps.

First of all, we construct a sort that is called *nativevocsystem* (resp. *foreignvocsystem*). This is nothing else but a list of native (resp. *foreign*) words, extracted from a given vocubulary system. Apart from the constructors for this sort and some error handling, an operation *GenerateNativeVocSystem* is provided, with arity *vocsystem* $\times$ *int*. This operation is used to generate a *nativevocsystem* with n entries, where n is the given integer. This operation will later on be used for the generation of a test suite with a user defined number of questions. The other operation which is of importance for the generation of the test suite is the operation *CountMaximumNativeScore*. This operation counts the total amount of possible translations for every word in the *nativevocsystem* and adds these numbers together to get the final maximal score. These sorts with their operations are specified in *PreTestVocSystemSpec*.

spec PreTestVocSystemSpec

freely generated type NativeVocSystem

import DictionarySpec

import ListOfListOfItemsSpec

import IntSpec

opns

constr NewNativeVocSytem: vocsystem→ nativevocsytem

constr AddEntryToNativeVocSystem: word×nativevocsystem→ nativevocsystem

constr ERROR_NativeVocExceedsVoc: → nativevocsystem

constr ErrorNoEntriesLeft: → nativevocsystem

_In_: word×nativevocsystem→ boolean

AddToNativeVoc: word×nativevocsystem→ nativevocsystem

DeleteNativeEntry: word×nativevocsystem→ nativevocsystem

GetVocSystem: nativevocsystem→ vocsystem

RecGenerateNativeVocSystem: nativevocsystem×int×vocsystem→ nativevocsystem

GenerateNativeVocSystem: vocsystem×int→ nativevocsystem

GetNativeHead: nativevocsystem→ word

GetNativeTail: nativevocsystem→ nativevocsystem

MaxNativeScore: nativevocsystem→ nat

GetMaxNative: nativevocsystem×word→ word

GetNativeTranslationList: nativevocsystem×listoflistofitems→ listoflistofitems

GetOrderedNativeWordList: nativevocsystem×nativevocsystem→ nativevocsystem

GetNativeOrderedItemList: nativevocsystem×listoflistofitems→ listoflistofitems

GetListOfNativeWords: vocsystem×vocsystem→ nativevocsystem

constr NewForeignVocSytem: vocsystem→ foreignvocsytem

constr AddEntryToForeignVocSystem: word×foreignvocsystem→ foreignvocsystem

constr ERROR_ForeignVocExceedsVoc: → foreignvocsystem

constr ErrorNoEntriesLeft: → foreignvocsystem

_In_: word×foreignvocsystem→ boolean

AddToForeignVoc: word×foreignvocsystem→ foreignvocsystem

DeleteForeignEntry: word×foreignvocsystem→ foreignvocsystem

GetVocSystem: foreignvocsystem→ vocsystem

RecGenerateForeignVocSystem: foreignvocsystem×int×vocsystem→ foreignvocsystem

GenerateForeignVocSystem: vocsystem×int→ foreignvocsystem

GetForeignHead: foreignvocsystem→ word

GetForeignTail: foreignvocsystem→ foreignvocsystem

MaxForeignScore: foreignvocsystem→ nat

GetMaxForeign: foreignvocsystem×word→ word

GetForeignTranslationList: foreignvocsystem×listoflistofitems→ listoflistofitems

GetOrderedForeignWordList: foreignvocsystem×foreignvocsystem→ foreignvocsystem

GetForeignOrderedItemList: foreignvocsystem×listoflistofitems→ listoflistofitems

GetListOfForeignWords: vocsystem×vocsystem→ foreignvocsystem

vars

ntvoc:nativevocsystem

size:int

voc:vocsystem

it:vocitem

itlst: listofitems

w,v,output:word

lstlstit: listoflistofitems

l1,l2: language

axioms

GetVocSystem(NewNativeVocSystem(voc)) = voc

GetVocSystem(AddEntryToNativeVocSystem(w,ntvoc))=
GetVocSystem(ntvoc)

DeleteNativeEntry(w,NewNativeVocSystem(voc)) =
NewNativeVocSystem(voc)

DeleteNativeEntry(w,AddEntryToNativeVocSystem(v,ntvoc)) =
If w = v **then** ntvoc
else AddEntryToNativeVocSystem(v,DeleteNativeEntry(w,ntvoc))

AddToNativeVoc(w,ntvoc)=
If (w In ntvoc) **then** ntvoc
else AddEntryToNativeVocSystem(w,ntvoc)

(w In NewNativeVocSystem(voc)) = False

(w In AddEntryToNativeVocSystem(v,nativevoc)) =
If w = v **then** True **else** (w In ntvoc)

RecGenerateNativeVocSystem(ntvoc,size,voc) =
Let vcitm := RandomVocItem(voc)
if vcitm = ERROR_OutOfEntries
then
If size=0 **then** ntvoc **else** ERROR_NativeVocExcedsVoc
else
If size=0 **then** ntvoc **else**

```
    Let nextntvoc :=
      SetNativeFlags(CorrectWord(GetNative(vcitm)),voc)
    RecGenerateNativeVocSystem(
      AddEntryToNativeVocSystem(
        CorrectWord(GetNative(GetPair(vcitm))),ntvocsys),
      size-1,nextntvoc)
GenerateNativeVocSystem(voc,size) =
  RecGenerateNativeVocSystem(NewNativeVocSystem(voc),size,voc)
GetNativeHead(NewNativeVocSystem(voc))=ERROR_NoWordLeft
GetNativeHead(AddEntryToNativeVocSystem(w,ntvoc)=w
GetNativeTail(NewNativeVocSystem(voc))=ERROR_NoEntriesLeft
GetNativeTail(AddEntryToNativeVocSystem(w,ntvoc)=ntvoc
MaxNativeScore(NewNativeVocSystem(voc)) = 0
MaxNativeScore(AddEntryToNativeVocSystem(w,ntvoc)) =
  Let voc := GetVocSystem(ntvoc)
  MaxNativeScore(ntvoc) +
  Length(TranslateIntoForeign(GetString(w),voc))
GetMaxNative(NewNativeVocSystem(voc),output) = output
GetMaxNative(
  AddEntryToNativeVocSystem(w,NewNativeVocSystem(voc)),
  output) =
  If output=DummyValue then w else output
GetMaxNative(ntvoc,output) =
  Let first := GetNativeHead(ntvoc)
  Let second := GetNativeHead(GetNativeTail(ntvoc))
  If not GetNativeHead(ntvoc) = ErrorNoWordLeft and
     not GetNativeTail(GetNativeTail(ntvoc)) = ErrorNoEntriesLeft
    If IsStringSmaller(GetString(first),GetString(second)) then
      GetMaxNative(GetNativeTail(ntvoc))
    else
      GetMaxNative(
        AddEntryToNativeVocSystem(first,GetNativeTail(ntvoc)),
        first)
GetOrderedNativeWordList(ntvoc,ntout) =
  If GetNativeHead(ntvoc) == ErrorNoWordLeft then ntout
  else
    Let max := GetMaxNative(ntvoc,DummyValue)
    GetOrderedNativeWordList(DeleteNativeEntry(max,ntvoc)),
    AddEntryToNativeVocSystem(max,ntout))
GetNativeTranslationList(NewNativeVocSystem(voc),lstlstit) =
  lstlstit
GetNativeTranslationList(
  AddEntryToNativeVocSystem(w,ntvoc),lstlistit) =
    GetNativeTranslationList(ntvoc,
      AddToItemListsList(
        TranslateIntoForeign(GetString(w), GetVocSystem(ntvoc)),
```

```
      lstlstit))
GetNativeOrderedItemList(nativevoc,lstlstit) =
  GetNativeTranslationList(
    GetOrderedNativeWordList(nativevoc,
      NewNativeVocSystem(GetVocSystem(nativevoc))),lstlstit)
GetListOfNativeWords(Dictionary(l1,l2,[ ]),voc) =
  NativeVocSystem(voc)
GetListOfNativeWords(Dictionary(l1,l2,NewItem(it,itlst)),voc) =
 AddToNativeVoc(
   CorrectWord(GetNative(GetPair(it))),
   GetListOfNativeWords(Dictionary(l1,l2,itlst),voc))
```

The axioms of the operations for the foreign case are analogeous to the native case.

endspec

These sorts, *nativevocsystem* and *foreignvocsystem*, are then used to construct the actual test suite. This test suite is specified in the sorts *testnativevocsystem* and *testforeignvocsystem*, and is actually nothing else than a *nativevocsystem* (resp. a *foreignvocsystem*) and an integer, where this integer represents the actual score of the user. This sort is seen in *TestVocSystemSpec*.

spec TestVocSystemSpec

opns

constr GenerateNativeTest: nativevocsystem $\times$ Int $\rightarrow$ testnativevocsystem

GetNativeWord: testnativevocsystem $\rightarrow$ word

GetMaxNativeScore: testnativevocsystem $\rightarrow$ Nat

GetNativeScore: testnativevocsystem $\rightarrow$ Int

GetNativeVocSystem: testnativevocsystem $\rightarrow$ nativevocsytem

AdaptNativeVocSysyetm: string $\times$ testnativevocsystem $\rightarrow$ testnativevocsystem

GetnativeStringList: listofitems $\times$ listofstrings $\rightarrow$ listofstrings

constr GenerateForeignTest: foreignvocsystem $\times$ Int $\rightarrow$ testforeignvocsystem

GetForeignWord: testforeignvocsystem $\rightarrow$ word

GetMaxForeignScore: testforeignvocsystem $\rightarrow$ Nat

GetForeignScore: testforeignvocsystem $\rightarrow$ Int

GetForeignVocSystem: testforeignvocsystem $\rightarrow$ foreignvocsytem

AdaptForeignVocSystem: string $\times$ testforeignvocsystem $\rightarrow$ testforeignvocsystem

GetforeignStringList: listofitems $\times$ listofstrings $\rightarrow$ listofstring

GetPairList: listofitems $\times$ pairlist $\rightarrow$ pairlist

GetListofPairLists: listoflistofites $\times$ listofpairlists $\rightarrow$ listofpairlists

vars

tnvoc : testnativevocsystem

tfvoc : testforeignvocsystem

fvoc : foreignvocsystem

```
        pos : Nat
        score : Int
        str : string
        vit : vocitem
        ilst : listofitems
        illst : listoflistofitems
        sllst : listofpairlists
        slst : listofstrings
        plst : pairlist
    axioms
        GetNativeVocSystem(GenerateNativeTest(nvoc,score)) = nvoc
        GetMaxNativeScore(GenerateNativeTest(nvoc,score)) = MaxNativeScore(nvoc)
        GetNativeWord(tnvoc) = GetNativeHead(GetNativeVocSystem(tnvoc))
        AdaptNativeVocSystem(str,tnvoc) =
          if MakeWord(str,GetForeignLanguage(
              GetVocSystem(GetNativeVocSystem(tnvoc)))) = ErrorMispelled
            or not MakeItem(MakePair(
              GetNativeWord(tnvoc),
              MakeWord(str,GetForeignLanguage(
                GetVocSystem(GetNativeVocSystem(tnvoc))))),false)
            ∈ GetItemList(GetVocSystem(GetNativeVocSystem(tnvoc))) then
            GenerateNativeTest(GetNativeVocSystem(tnvoc),
            GetNativeScore(tnvoc)-1)
          else
            GenerateNativeTest(GetNativeVocSystem(tnvoc),GetNativeScore(tnvoc)+1)
        GetNativeStringList([ ],slst) = slst
        GetNativeStringList(NewItem(vit,ilst),slst) =
          GetNativeStringList(ilst, AddString(GetNative(GetPair(vit)),slst))
        GetPairList([ ],plst) = plst
        GetPairList(NewItem(vit,ilst),plst) =
          GetPairList(ilst,NewPair(GetPair(vit),plst))
        GetListofPairLists([ ],sllst) = sllst
        GetListofPairLists(NewItemListsList(ilst,illst),sllst) =
          GetListOfPairLists(illst, NewPairList(GetPairList(ilst,[ ]),sllst))
        The axioms of the operations for the foreign case are analogeous to the native
        case.
endspec
```

2.10 Doors and Windows: the User Interface

Now that we have all the operations and sorts we need, we can provide an easy interface to all these operations, where the user only has to think about the basic aspects. These basic aspects are:

1. for the dictionary: the languages and the strings he uses to enter words;
2. for the test suite: the number of questions he wants to be asked

The fact that these are the only things the user now finally needs is reflected in the arity of the operations in the final user interface, described in the specification *UserInterfaceSpec*. This level of the student dictionary system was our final goal, since at this point the user only has a limited amount of access to the internal structure of the student dictionary system. This means for instance he can see his score but cannot manually modify it (as if there were a window between the user and the score). He can for instance also only give answers in a list of strings, thus preventing him of giving the same answer multiple times (he is only allowed to enter the system through certain doors or entrances).

spec UserInterfaceSpec

freely generated type Score

opns

constr _Out_: Int×NaT→ Score

GenerateDictionary: language×language→ vocsystem
AddEntry: string×string×vocsystem→ vocsystem
DeleteEntry: string×string×vocsystem→ vocsystem
GiveNativeTranslation: string×vocsystem→ listofstrings
GiveForeignTranslation: string×vocsystem→ listofstrings
GetAllTranslationsByNativeOrder: vocsystem→ listofpairlists
GetAllTranslationsByForeignOrder: vocsystem→ listofpairlists
TestMyNativeKnowledge: Nat×vocsystem→ testnativevocsystem
TestMyForeignKnowledge: Nat×vocsystem→ testforeignvocsystem
GiveNativeAnswers: listofstrings×testforeignvocsystem→ testforeignvocsystem
GiveForeignAnswers: listofstrings×testnativevocsystem→ testnativevocsystem
ShowMyScoreInNative: testnativevocsystem×testnativevocsystem→ Score
ShowMyScoreInForeign: testforeignvocsystem×testforeignvocsystem→ Score

vars

l1, l2: language

s1, s2: string
lst: listofstrings
sdic: vocsystem
no: Nat

tnvoc,nansw: testnativevocsystem
tfvoc,fansw: testforeignvocsystem

axioms

GenerateDictionary(l1,l2) = Dictionary(l1,l2,[])

AddEntry(s1,s2,sdic) = AddEntryToDictionary(s1,s2,sdic)
DeleteEntry(s1,s2,sdic) = DeleteEntryFromDictionary(s1,s2,sdic)
GiveNativeTranslation(s1,sdic) = GetNativeStringList(TranslateIntoNative(s1,sdic),[])
GiveForeignTranslation(s1,sdic) = GetForeignStringList(TranslateIntoForeign(s1,sdic),[])
GetAllTranslationsByNativeOrder(sdic) =
  GetListOfPairLists(GetNativeOrderedItemList(
    GetListOfNativeWords(sdic,sdic),[]),[])
GetAllTranslationsByForeignOrder(sdic) =
  GetListOfPairLists(GetForeignOrderedItemList(
    GetListOfForeignWords(sdic,sdic),[]),[])
TestMyNativeKnowledge(no,sdic) =
  GenerateNativeTest(GenerateNativeVocSystem(sdic,no),0)
TestMyForeignKnowledge(no,sdic) =
  GenerateForeignTest(GenerateForeignVocSystem(sdic,no),0)
GiveNativeAnswers([],tfvoc) = DeleteForeignEntry(GetForeignHead(tfvoc),tfvoc)
GiveNativeAnswers(NewString(s1,lst),tfvoc) =
  GiveNativeAnswers(lst,AdaptForeignVocSystem(s1,tfvoc))
GiveForeignAnswers([],tnvoc) = DeleteNativeEntry(GetNativeHead(tnvoc),tnvoc)
GiveForeignAnswers(NewString(s1,lst),tnvoc) =
  GiveForeignAnswers(lst,AdaptNativeVocSystem(s1,tnvoc))
ShowMyScoreInNative(nansw,tnvoc) = GetNativeScore(nansw) Out
GetMaxNativeScore(tnvoc)
ShowMyScoreInForeign(fansw,tfvoc) = GetForeignScore(fansw) Out
GetMaxForeignScore(tfvoc)

endspec

Note that the operations *NativeScore* and *GetForeignScore* take two *testnativevocsystems* resp. two testforeignvocsystems. The first entry is the test vocabulary system after all questions have been answered, and consists out of an empty native (or foreign) word vocabulary system and the user's final score. The second entry is the original test suite generated, and is used to get the maximal score.

Chapter 3

The CafeOBJ Implementation

We conclude this paper with the cafeOBJ implementation of the specification introduced in in the previous chapter. This chapter contains two sections: the first contains the cafeOBJ implementation itself, and the second section contains an example of the way this cafeOBJ implementation and the specification work.

3.1 The cafeOBJ Modules

As we explained in the previous chapter, we divided the student dictionary system into two major parts: the first one containing all the tools needed to build a student dictionary system, and the second level containing the construction of the dictionary system itself.

The first cafeOBJ module shown below is called `lowlevel.mod` and contains the implementation of the low level material of the dictionary as depicted in Figure 2.1.

```
** -----------------------------------------------------
**   Specification of the type list
** -----------------------------------------------------


module! ElementSpec
{
  [element]
}

module! SimpleListSpec( E :: ElementSpec )
{

  imports { protecting (NAT) }

  signature {
    [list]
```

```
    op [] : -> list { constr }
    op AddToSimpleList : element.E list -> list { constr }
    ** op [_] : element.E -> list
    op Head : list -> element.E
    op Tail : list -> list
    op _+_ : list list -> list
    op _in_ : element.E list -> Bool
    op Length : list -> Nat
  }
  axioms {
    vars l m n : list
    vars f e : element.E

    eq Head(AddToSimpleList(e,l)) = e .
    eq Tail(AddToSimpleList(e,l)) = l .

    ** eq [e] = AddToSimpleList(e,[]) .

    eq [] + l = l .
    eq AddToSimpleList(e,l) + m = AddToSimpleList(e,l + m) .

    eq (e in []) = false .
    ceq (e in AddToSimpleList(f, l)) = true if (e == f) .
    ceq (e in AddToSimpleList(f, l)) = (e in l) if not(e == f) .

    eq Length([]) = 0 .
    eq Length(AddToSimpleList(e,l)) = Length(l) + 1 .
  }
}


module! ListSpec
{
  protecting (SimpleListSpec)

  signature {
    op AddToList : element.E list -> list
  }

 axioms {
   vars e : element.E
   vars l : list

   ceq AddToList(e,l) = AddToSimpleList(e,l) if not(e in l) .
   ceq AddToList(e,l) = l if (e in l) .
  }
}
```

```
** ----------------------------------------------------------
**  Specification of the type charorder
** ----------------------------------------------------------

module! CharacterOrderSpec
{
  protecting (ListSpec(CHARACTER { sort element -> Character }) *
  {
    sort list -> charorder ,
    op AddToList -> AddToOrder ,
    op AddToSimpleList -> NewChar ,
    op [] -> coEmpty
  })
}

** -------------------------------------------------------------------
**   Specification of the type String
**   (StringSpec extends the cafeObj specification of String, i.e.
**   the lexicographical order for strings is taken from cafeObj.)
** -------------------------------------------------------------------

module! StringSpec
{
  imports {
    protecting (STRING)
    protecting (NAT)
    protecting (CharacterOrderSpec)
  }

  signature {
    op Tail : String -> String
    op Head : String -> Character
    op CheckAlphabet : String charorder -> Bool
    op CheckAlphabetInternal : String charorder -> Bool
    op IsStringSmaller : String String -> Bool

    op ErrorNoStringLeft : -> String { constr }
 }

  axioms {
    vars s1 s2 : String
    vars co : charorder

    eq Head(s1) = schar(s1,0) .
    eq Tail(s1) = substring(s1,1) .
```

```
    eq IsStringSmaller(s1,s2) = string<(s1,s2) .

    ceq CheckAlphabet(s1,co) = false if length(s1) == 0 .
    ceq CheckAlphabet(s1,co) = (Head(s1) in co) if length(s1) == 1 .
    ceq CheckAlphabet(s1,co) = CheckAlphabetInternal(s1,co) if length(s1) > 1 .

    ceq CheckAlphabetInternal(s1,co) = false if not (Head(s1) in co) .
    ceq CheckAlphabetInternal(s1,co) = CheckAlphabet(Tail(s1),co) if Head(s1) in co .


  }
}


** -------------------------------------------------------------------
**   Specification of the type rules
**   (In the cafeObj implementation there is the following restriction
**   according to our loose specification: a rule must consist of
**   exactly two characters.
** -------------------------------------------------------------------

module! SimpleRuleSpec
{
  protecting (ListSpec(StringSpec { sort element -> String }) *
  {
    sort list -> rules ,
    op AddToSimpleList -> NewRule ,
    op [] -> rEmpty,
    op AddToList -> AddFlexibleRule
  })
}

module! RuleSpec
{
  imports {
    protecting (SimpleRuleSpec)
    protecting (BOOL)
    protecting (NAT)
  }

  signature {

    op AddToRules : String rules -> rules
    op MatchRule : String rules -> Bool

    op FindPatternIntern : String String -> Bool
```

```
    op IsPatternIntern : String String -> Bool

  }

  axioms {
    vars str r : String
    vars R : rules

    ceq AddToRules(str,R) = AddFlexibleRule(str,R) if length(str) == s(1) .
    ceq AddToRules(str,R) = R if not length(str) == s(1) .

    eq MatchRule(str,rEmpty) = false .
    ceq MatchRule(str,NewRule(r,R)) = true if IsPatternIntern(str,r) == true .
    ceq MatchRule(str,NewRule(r,R)) = MatchRule(str,R)
          if not IsPatternIntern(str,r) == true .

    ceq IsPatternIntern(str,r) = false if length(str) <= 1 .
    ceq IsPatternIntern(str,r) = FindPatternIntern(str,r) if length(str) > 1 .

    ceq FindPatternIntern(str,r) = true
         if Head(str) == Head(r) and Head(Tail(str)) == Head(Tail(r)) .
    ceq FindPatternIntern(str,r) = IsPatternIntern(Tail(str),r)
         if not (Head(str) == Head(r) and Head(Tail(str)) == Head(Tail(r))) .

  }
}

** ------------------------------------------------------------
**   Specification of the type language
** ------------------------------------------------------------

module! LanguageSpec
{
  imports {
    protecting (CharacterOrderSpec)
    protecting (RuleSpec)
  }

  signature {
    [language]

    op MakeLanguage : rules charorder -> language
    op GetRules : language -> rules
    op GetCharacterOrder : language -> charorder
  }

  axioms {
```

```
    vars co : charorder
    vars r : rules

    eq GetRules(MakeLanguage(r,co)) = r .
    eq GetCharacterOrder(MakeLanguage(r,co)) = co .
  }
}


** ------------------------------------------------------------
**   Specification of the type word
** ------------------------------------------------------------

module! WordSpec
{
  imports {
    protecting (StringSpec)
    protecting (LanguageSpec)
  }

  signature {
    [word]

    op MakeWord : String language -> word
    op GetString : word -> String

    op ErrorMisspelled : -> word { constr }
    op ErrorNoWordLeft : -> word { constr }
    op DummyValue : -> word { constr }

    op CorrectWord : String -> word { constr }
  }

  axioms {
    vars s : String
    vars l : language

    eq GetString(CorrectWord(s)) = s .
    eq GetString(ErrorNoWordLeft) = ErrorNoStringLeft .

    ceq MakeWord(s,l) = CorrectWord(s)
          if (not MatchRule(s,GetRules(l))) and CheckAlphabet(s,GetCharacterOrder(l)) .

    ceq MakeWord(s,l) = ErrorMisspelled
          if not ((not MatchRule(s,GetRules(l))) and CheckAlphabet(s,GetCharacterOrder(l)
  }
}
```

```
** -------------------------------------------------------------------
**  Specification of the type pair, pairlist, and listofpairlist
** -------------------------------------------------------------------

module! PairSpec
{
  imports {
    protecting (StringSpec)
    protecting (WordSpec)
  }

  signature {
    [pair]
    op MakePair : word word -> pair { constr }
    op GetNative : pair -> String
    op GetForeign : pair -> String

    op CheckWords : word word -> Bool
  }

  axioms {
    vars n f : word


    ceq CheckWords(n,f) = true
          if not n == ErrorMisspelled and not f == ErrorMisspelled .
    ceq CheckWords(n,f) = false
          if n == ErrorMisspelled or f == ErrorMisspelled .


    ceq GetNative(MakePair(n,f)) = GetString(n) if CheckWords(n,f) .
    ceq GetForeign(MakePair(n,f)) = GetString(f) if CheckWords(n,f) .

  }
}

module! PairListSpec
{
  protecting (ListSpec(PairSpec { sort element -> pair }) *
  {
    sort list -> pairlist ,
    op AddToList -> AddPair ,
    op AddToSimpleList -> NewPair,
    op [] -> plEmpty
```

```
  })
}

module! ListOfPairListsSpec
{
  protecting (ListSpec(PairListSpec { sort element -> pairlist }) *
  {
    sort list -> listofpairlists ,
    op AddToList -> AddPairList ,
    op [] -> lplEmpty,
    op AddToSimpleList -> NewPairList
  })
}


** -----------------------------------------------------
**   Specification of the type vocitem and listofitems
** -----------------------------------------------------

module! VocItemSpec
{
  imports {
    protecting (PairSpec)
  }

  signature {
    [vocitem]
    op MakeItem : pair Bool -> vocitem
    op GetPair : vocitem -> pair
    op GetFlag : vocitem -> Bool
    op ErrorOutOfEntries : -> vocitem
  }

  axioms {
    vars vi : vocitem
    vars p : pair
    vars f : Bool

    eq GetFlag(MakeItem(p,f)) = f .
    eq GetPair(MakeItem(p,f)) = p .
  }
}


module! ListOfItemsSpec
{
  protecting (ListSpec(VocItemSpec { sort element -> vocitem }) *
```

```
  {
    sort list -> listofitems ,
    op AddToList -> AddItem ,
    op [] -> liEmpty,
    op AddToSimpleList -> NewItem ,
  })
}

** --------------------------------------
** Specification of the ListOfStringsSpec
** (needed for the user interface when the
**  user enters translations in the test
**  suite).
** --------------------------------------

module! ListOfStringsSpec
{
  protecting (ListSpec(StringSpec { sort element -> String }) *
  {
    sort list -> listofstrings ,
    op AddToList -> AddString ,
    op [] -> lsEmpty,
    op AddToSimpleList -> NewString
  })
}

module ListOfStringListSpec
{
  protecting (ListSpec(ListOfStringsSpec { sort element -> listofstrings }) *
  {
    sort list -> listofstringlist ,
    op AddToList -> AddStringList ,
    op [] -> lslEmpty,
    op AddToSimpleList -> NewStringList
  })
}

module! ListOfListOfItemsSpec
{
  protecting (ListSpec(ListOfItemsSpec { sort element -> listofitems }) *
  {
    sort list -> listoflistofitems ,
    op [] -> lliEmpty,
    op AddToList -> AddToItemListsList ,
    op AddToSimpleList -> NewItemListsList
  })
}
```

The module `dictionary.mod` contains the upper level op the specification, ultimately leading to the implementation of the user interface part of the specification.

```
** Import low-level stuff:
input lowlevel.mod

** ------------------------------------------------
** Specification of the Vocabulary System sort
** ------------------------------------------------

module! DictionarySpec
{
  imports {
    protecting (ListOfItemsSpec)
  }

  signature {
    [vocsystem]

    ** The Constructors of the sort:
    op Dictionary : language language listofitems -> vocsystem     { constr }
    op ErrorNativeWordIsMisspelled : -> vocsystem                  { constr }
    op ErrorForeignWordIsMisspelled : -> vocsystem                 { constr }

    op AddEntryToDictionary : String String vocsystem -> vocsystem
    op DeleteEntryFromDictionary : String String vocsystem -> vocsystem

    ** The other operations:
    op GetNativeLanguage : vocsystem -> language
    op GetForeignLanguage : vocsystem -> language
    op GetItemList : vocsystem -> listofitems
    op RandomVocItem : vocsystem -> vocitem
    op ResetFlags : vocsystem -> vocsystem
    op ResetFlagsRec : listofitems listofitems -> listofitems
    op SetVocItemFlag : vocitem Bool -> vocitem
    op SetNativeFlags : word vocsystem -> vocsystem
    op SetNativeFlagsRec : word listofitems listofitems -> listofitems
    op SetForeignFlags : word vocsystem -> vocsystem
    op SetForeignFlagsRec : word listofitems listofitems -> listofitems

    ** We already provide the translation operations:
    op TranslateIntoNative : String vocsystem -> listofitems
    op TranslateIntoForeign : String vocsystem -> listofitems
```

```
  op TranslateIntoNativeRec : String listofitems listofitems -> listofitems
  op TranslateIntoForeignRec : String listofitems listofitems -> listofitems
  }

axioms {
  vars l1 l2 : language
  vars  il o : listofitems
  var  p     : pair
  vars f b   : Bool
  var  voc   : vocsystem
             vars s1 s2 : String
  var  w     : word
  var  vi    : vocitem

  ** We start out with some nice operations:
  eq GetNativeLanguage(Dictionary(l1,l2,il)) = l1 .
  eq GetForeignLanguage(Dictionary(l1,l2,il)) = l2 .
  eq GetItemList(Dictionary(l1,l2,il)) = il .

  ** The next operation is the addition of elements to the dictionary:
  ** We first check whether the native word is correctly spelled:
  ceq AddEntryToDictionary(s1,s2,voc) = ErrorNativeWordIsMisspelled
    if MakeWord(s1,GetNativeLanguage(voc)) == ErrorMisspelled .
  ** If this is the case, then we check the foreign spelling:
  ceq AddEntryToDictionary(s1,s2,voc) = ErrorForeignWordIsMisspelled
    if not MakeWord(s1,GetNativeLanguage(voc)) == ErrorMisspelled
       and MakeWord(s2,GetForeignLanguage(voc)) == ErrorMisspelled .
  ** If both are correct, we continue:
  ceq AddEntryToDictionary(s1,s2,voc) =
    Dictionary(GetNativeLanguage(voc),GetForeignLanguage(voc),
    AddItem(
      MakeItem(
        MakePair(
          MakeWord(s1,GetNativeLanguage(voc)),
          MakeWord(s2,GetForeignLanguage(voc))),
        false),
      GetItemList(voc)))
      if not MakeWord(s1,GetNativeLanguage(voc)) == ErrorMisspelled
         and not MakeWord(s2,GetForeignLanguage(voc)) == ErrorMisspelled .

  ** We allow the user to remove entries from the dictionary as well:
  eq DeleteEntryFromDictionary(s1,s2,Dictionary(l1,l2,liEmpty)) =
      Dictionary(l1,l2,liEmpty) .
  ceq DeleteEntryFromDictionary(s1,s2,Dictionary(l1,l2,NewItem(vi,il))) =
      Dictionary(l1,l2,il)
    if s1 == GetNative(GetPair(vi)) and s2 == GetForeign(GetPair(vi)) .
  ceq DeleteEntryFromDictionary(s1,s2,Dictionary(l1,l2,NewItem(vi,il))) =
```

```
    AddEntryToDictionary(GetNative(GetPair(vi)),
                         GetForeign(GetPair(vi)),
                         DeleteEntryFromDictionary(
                            s1,s2,Dictionary(l1,l2,il)))
  if not (s1 == GetNative(GetPair(vi)) and s2 == GetForeign(GetPair(vi))) .

** Let us now provide the operations on the flags:
eq SetVocItemFlag(vi,b) = MakeItem(GetPair(vi),b) .
** Flags are set recursively:
ceq SetNativeFlagsRec(w,il,o) = o if il == liEmpty .
ceq SetNativeFlagsRec(w,il,o) =
  SetNativeFlagsRec(w,Tail(il),AddItem(SetVocItemFlag(Head(il),true),o))
               if not il == liEmpty
                  and GetNative(GetPair(Head(il))) == GetString(w) .
ceq SetNativeFlagsRec(w,il,o) =
  SetNativeFlagsRec(w,Tail(il),AddItem(Head(il),o))
   if not il == liEmpty
      and not GetNative(GetPair(Head(il))) == GetString(w) .

ceq SetForeignFlagsRec(w,il,o) = o if il == liEmpty .
ceq SetForeignFlagsRec(w,il,o) =
  SetForeignFlagsRec(w,Tail(il),AddItem(
                                  SetVocItemFlag(Head(il),true),o))
               if not il == liEmpty
                  and GetForeign(GetPair(Head(il))) == GetString(w) .
ceq SetForeignFlagsRec(w,il,o) = SetForeignFlagsRec(w,Tail(il),AddItem(
                                  Head(il),o))
               if not il == liEmpty
                  and not GetForeign(GetPair(Head(il))) == GetString(w) .

** The recursion is always called starting with empty output:
eq SetNativeFlags(w,voc) = Dictionary(
  GetNativeLanguage(voc),
  GetForeignLanguage(voc),
  SetNativeFlagsRec(w,GetItemList(voc),liEmpty)) .
eq SetForeignFlags(w,voc) = Dictionary(
  GetNativeLanguage(voc),
  GetForeignLanguage(voc),
  SetForeignFlagsRec(w,GetItemList(voc),liEmpty)) .
** We also need to reset the flags when making a new test suite:
ceq ResetFlagsRec(il,o) = o if il == liEmpty .
ceq ResetFlagsRec(il,o) = ResetFlagsRec(Tail(il),
  AddItem(SetVocItemFlag(Head(il),false),o)) if not il == liEmpty .
eq ResetFlags(voc) = Dictionary(GetNativeLanguage(voc),
  GetForeignLanguage(voc),ResetFlagsRec(GetItemList(voc),liEmpty)) .

** The only thing missing is the random operation. It is not easy to
```

```
    ** implement a pseudo random function in cafeOBJ, so instead we simply
    ** walk through the list and take the first vocabulary item that is
    ** eligible.
    ceq RandomVocItem(voc) = ErrorOutOfEntries
      if GetItemList(voc) == liEmpty .
    ceq RandomVocItem(voc) = Head(GetItemList(voc))
      if not GetItemList(voc) == liEmpty and
         not GetFlag(Head(GetItemList(voc))) .
    ceq RandomVocItem(voc) = RandomVocItem(Dictionary(GetNativeLanguage(voc),
      GetForeignLanguage(voc),Tail(GetItemList(voc))))
      if not GetItemList(voc) == liEmpty and
      GetFlag(Head(GetItemList(voc))) .

    ** Now it is time to implement the translation operations, as they will be
    ** used in the nativevocsystem sort.
    ceq TranslateIntoNativeRec(s1,il,o) = o if il == liEmpty .
    ceq TranslateIntoNativeRec(s1,il,o) = TranslateIntoNativeRec(s1,Tail(il),
          AddItem(Head(il),o))
            if not il == liEmpty and GetForeign(GetPair(Head(il))) == s1 .
    ceq TranslateIntoNativeRec(s1,il,o) = TranslateIntoNativeRec(s1,Tail(il),o)
          if not il == liEmpty and not GetForeign(GetPair(Head(il))) == s1 .
    eq TranslateIntoNative(s1,voc) = TranslateIntoNativeRec(s1,
                                      GetItemList(voc),liEmpty) .

    ceq TranslateIntoForeignRec(s1,il,o) = o if il == liEmpty .
    ceq TranslateIntoForeignRec(s1,il,o) = TranslateIntoForeignRec(s1,Tail(il),
          AddItem(Head(il),o))
            if not il == liEmpty and GetNative(GetPair(Head(il))) == s1 .
    ceq TranslateIntoForeignRec(s1,il,o) = TranslateIntoForeignRec(s1,
                                             Tail(il),o)
          if not il == liEmpty and not GetNative(GetPair(Head(il))) == s1 .
    eq TranslateIntoForeign(s1,voc) = TranslateIntoForeignRec(s1,
                                      GetItemList(voc),liEmpty) .
    }
}

** ----------------------------------------------------
** Specification of the Native Vocabulary System sort
** ----------------------------------------------------

module! PreTestVocSystemSpec
{
  imports {
    protecting (DictionarySpec)
    protecting (ListOfListOfItemsSpec)
    protecting (INT)
    }
```

```
signature {
  [ nativevocsystem
    foreignvocsystem ]

  ** ----------------
  ** The Native Part:
  ** ----------------
  ** The constructors of the sort:
  op NewNativeVocSystem : vocsystem -> nativevocsystem { constr }
  op AddEntryToNativeVocSystem : word nativevocsystem
                               -> nativevocsystem { constr }
  op ErrorNativeVocExceedsVoc : -> nativevocsystem { constr }
  op ErrorNoEntriesLeft : -> nativevocsystem { constr }

  ** The other operations:
  op _In_ : word nativevocsystem -> Bool
  op AddToNativeVoc : word nativevocsystem -> nativevocsystem
  op DeleteNativeEntry : word nativevocsystem -> nativevocsystem

  op GetVocSystem : nativevocsystem -> vocsystem
  op RecGenerateNativeVocSystem : nativevocsystem Int vocsystem
                                -> nativevocsystem
  op GenerateNativeVocSystem : vocsystem Int -> nativevocsystem
  op GetNativeHead : nativevocsystem -> word
  op GetNativeTail : nativevocsystem -> nativevocsystem
  op MaxNativeScore : nativevocsystem -> Nat
  op GetMaxNative : nativevocsystem word -> word
  op GetNativeTranslationList : nativevocsystem listoflistofitems
                              -> listoflistofitems
  op GetOrderedNativeWordList : nativevocsystem nativevocsystem
                              -> nativevocsystem
  op GetNativeOrderedItemList : nativevocsystem listoflistofitems
                              -> listoflistofitems
  op GetListOfNativeWords : vocsystem vocsystem -> nativevocsystem

  ** -----------------
  ** The Foreign Part:
  ** -----------------
  ** The constructors of the sort:
  op NewForeignVocSystem : vocsystem -> foreignvocsystem { constr }
  op AddEntryToForeignVocSystem : word foreignvocsystem
                                -> foreignvocsystem { constr }
  op ErrorForeignVocExceedsVoc : -> foreignvocsystem { constr }
  op ErrorNoForeignEntriesLeft : -> foreignvocsystem { constr }

  ** The other operations:
```

```
  op _InForeign_ : word foreignvocsystem -> Bool
  op AddToForeignVoc : word foreignvocsystem -> foreignvocsystem
  op DeleteForeignEntry : word foreignvocsystem -> foreignvocsystem

  op GetVocSystem : foreignvocsystem -> vocsystem
  op RecGenerateForeignVocSystem : foreignvocsystem Int vocsystem
                                   -> foreignvocsystem
  op GenerateForeignVocSystem : vocsystem Int -> foreignvocsystem
  op GetForeignHead : foreignvocsystem -> word
  op GetForeignTail : foreignvocsystem -> foreignvocsystem
  op MaxForeignScore : foreignvocsystem -> Nat
  op GetMaxForeign : foreignvocsystem word -> word
  op GetForeignTranslationList : foreignvocsystem listoflistofitems
                                 -> listoflistofitems
  op GetOrderedForeignWordList : foreignvocsystem foreignvocsystem
                                 -> foreignvocsystem
  op GetForeignOrderedItemList : foreignvocsystem listoflistofitems
                                 -> listoflistofitems
  op GetListOfForeignWords : vocsystem vocsystem -> foreignvocsystem
}

axioms {
  var  nativevoc  : nativevocsystem
  var  natout     : nativevocsystem
  var  foreignvoc : foreignvocsystem
  var  forout     : foreignvocsystem
  var  size  : Int
  var  voc   : vocsystem
  vars w v   : word
  var  output     : word
  var  lst        : listoflistofitems
  var  it         : vocitem
  var  itlst      : listofitems
  vars l1 l2      : language

  ** ----------------
  ** The Native Part:
  ** ----------------
  eq GetVocSystem(NewNativeVocSystem(voc)) = voc .
  eq GetVocSystem(AddEntryToNativeVocSystem(w,nativevoc)) =
       GetVocSystem(nativevoc) .

   eq DeleteNativeEntry(w,NewNativeVocSystem(voc)) =
        NewNativeVocSystem(voc) .
  ceq DeleteNativeEntry(w,AddEntryToNativeVocSystem(v,nativevoc)) = nativevoc
    if v == w .
  ceq DeleteNativeEntry(w,AddEntryToNativeVocSystem(v,nativevoc)) =
```

```
      AddEntryToNativeVocSystem(v,DeleteNativeEntry(w,nativevoc))
  if not v == w .

ceq AddToNativeVoc(w,nativevoc) = nativevoc if w In nativevoc .
ceq AddToNativeVoc(w,nativevoc) = AddEntryToNativeVocSystem(w,nativevoc)
  if not w In nativevoc .

eq w In NewNativeVocSystem(voc) = false .
ceq w In AddEntryToNativeVocSystem(v,nativevoc) = true if w == v .
ceq w In AddEntryToNativeVocSystem(v,nativevoc) = ( w In nativevoc )
  if not w == v .

ceq RecGenerateNativeVocSystem(nativevoc,size,voc) =
      ErrorNativeVocExceedsVoc
  if RandomVocItem(voc) == ErrorOutOfEntries
  and not size == 0 .
ceq RecGenerateNativeVocSystem(nativevoc,size,voc) = nativevoc
  if not RandomVocItem(voc) == ErrorOutOfEntries
  and size == 0 .
ceq RecGenerateNativeVocSystem(nativevoc,size,voc) = nativevoc
  if RandomVocItem(voc) == ErrorOutOfEntries
  and size == 0 .
ceq RecGenerateNativeVocSystem(nativevoc,size,voc) =
  RecGenerateNativeVocSystem(
    AddEntryToNativeVocSystem(
      CorrectWord(GetNative(GetPair(RandomVocItem(voc)))),
      nativevoc),
      size - 1,
      SetNativeFlags(CorrectWord(GetNative(GetPair(RandomVocItem(voc)))),
      voc))
  if not RandomVocItem(voc) == ErrorOutOfEntries
  and not size == 0 .
eq GenerateNativeVocSystem(voc,size) = RecGenerateNativeVocSystem(
  NewNativeVocSystem(voc),size,voc) .

eq GetNativeHead(NewNativeVocSystem(voc)) = ErrorNoWordLeft .
eq GetNativeHead(AddEntryToNativeVocSystem(w,nativevoc)) = w .
eq GetNativeTail(NewNativeVocSystem(voc)) = ErrorNoEntriesLeft .
eq GetNativeTail(AddEntryToNativeVocSystem(w,nativevoc)) = nativevoc .

eq MaxNativeScore(NewNativeVocSystem(voc)) = 0 .
eq MaxNativeScore(AddEntryToNativeVocSystem(w,nativevoc)) =
  MaxNativeScore(nativevoc) + Length(
    TranslateIntoForeign(GetString(w),GetVocSystem(nativevoc))) .

eq GetMaxNative(NewNativeVocSystem(voc),output) = output .
ceq GetMaxNative(AddEntryToNativeVocSystem(w,NewNativeVocSystem(voc)),
```

```
                 output) = w
  if output == DummyValue .
ceq GetMaxNative(AddEntryToNativeVocSystem(w,NewNativeVocSystem(voc)),
                 output) = output
  if not output == DummyValue .
ceq GetMaxNative(nativevoc,output) =
      GetMaxNative(
  AddEntryToNativeVocSystem(GetNativeHead(nativevoc),
          GetNativeTail(GetNativeTail(nativevoc))),
  GetNativeHead(nativevoc))
  if not GetNativeHead(nativevoc) == ErrorNoWordLeft
     and not GetNativeTail(GetNativeTail(nativevoc)) == ErrorNoEntriesLeft
     and not IsStringSmaller(GetString(GetNativeHead(nativevoc)),
             GetString(GetNativeHead(GetNativeTail(nativevoc)))) .
ceq GetMaxNative(nativevoc,output) =
      GetMaxNative(GetNativeTail(nativevoc),GetNativeHead(
                                        GetNativeTail(nativevoc)))
  if not GetNativeHead(nativevoc) == ErrorNoWordLeft
     and not GetNativeTail(GetNativeTail(nativevoc)) == ErrorNoEntriesLeft
     and IsStringSmaller(GetString(GetNativeHead(nativevoc)),
             GetString(GetNativeHead(GetNativeTail(nativevoc)))) .
ceq GetOrderedNativeWordList(nativevoc,natout) = natout
  if GetNativeHead(nativevoc) == ErrorNoWordLeft .
ceq GetOrderedNativeWordList(nativevoc,natout) =
      GetOrderedNativeWordList(DeleteNativeEntry(
  GetMaxNative(nativevoc,DummyValue),nativevoc),
  AddEntryToNativeVocSystem(GetMaxNative(nativevoc,DummyValue),natout))
  if not GetNativeHead(nativevoc) == ErrorNoWordLeft .
eq GetNativeTranslationList(NewNativeVocSystem(voc),lst) = lst .
eq GetNativeTranslationList(AddEntryToNativeVocSystem(w,nativevoc),lst) =
     GetNativeTranslationList(nativevoc,
            AddToItemListsList(
             TranslateIntoForeign(GetString(w),
                GetVocSystem(nativevoc)),
             lst)) .
eq GetNativeOrderedItemList(nativevoc,lst) =
     GetNativeTranslationList(
       GetOrderedNativeWordList(nativevoc,
        NewNativeVocSystem(GetVocSystem(nativevoc))),
       lst) .

** The second vocabulary system is a copy of the original one and keeps all
** data that is lost from the first one as the first vocsystem is emptied
** in order to build the list of all native words .
eq GetListOfNativeWords(Dictionary(l1,l2,liEmpty),voc) =
     NewNativeVocSystem(voc) .
eq GetListOfNativeWords(Dictionary(l1,l2,NewItem(it,itlst)),voc) =
```

```
     AddToNativeVoc(
       CorrectWord(
        GetNative(GetPair(it))),
        GetListOfNativeWords(Dictionary(l1,l2,itlst),voc)) .

** -----------------
** The Foreign Part:
** -----------------
eq GetVocSystem(NewForeignVocSystem(voc)) = voc .
eq GetVocSystem(AddEntryToForeignVocSystem(w,foreignvoc)) =
     GetVocSystem(foreignvoc) .

 eq DeleteForeignEntry(w,NewForeignVocSystem(voc)) =
      NewForeignVocSystem(voc) .
ceq DeleteForeignEntry(w,AddEntryToForeignVocSystem(v,foreignvoc)) =
      foreignvoc
  if v == w .
ceq DeleteForeignEntry(w,AddEntryToForeignVocSystem(v,foreignvoc)) =
      AddEntryToForeignVocSystem(v,DeleteForeignEntry(w,foreignvoc))
  if not v == w .

ceq AddToForeignVoc(w,foreignvoc) = foreignvoc if w InForeign foreignvoc .
ceq AddToForeignVoc(w,foreignvoc) =
      AddEntryToForeignVocSystem(w,foreignvoc)
  if not w InForeign foreignvoc .

eq w InForeign NewForeignVocSystem(voc) = false .
ceq w InForeign AddEntryToForeignVocSystem(v,foreignvoc) = true
  if w == v .
ceq w InForeign AddEntryToForeignVocSystem(v,foreignvoc) =
        ( w InForeign foreignvoc )
  if not w == v .

ceq RecGenerateForeignVocSystem(foreignvoc,size,voc) =
     ErrorForeignVocExceedsVoc
  if RandomVocItem(voc) == ErrorOutOfEntries .
ceq RecGenerateForeignVocSystem(foreignvoc,size,voc) = foreignvoc
  if not RandomVocItem(voc) == ErrorOutOfEntries
  and size == 0 .
ceq RecGenerateForeignVocSystem(foreignvoc,size,voc) =
  RecGenerateForeignVocSystem(
    AddEntryToForeignVocSystem(
      CorrectWord(GetForeign(GetPair(RandomVocItem(voc)))),
      foreignvoc),
    size - 1,
    SetForeignFlags(
      CorrectWord(GetForeign(GetPair(RandomVocItem(voc)))),
```

```
    voc))
  if not RandomVocItem(voc) == ErrorOutOfEntries
  and not size == 0 .
eq GenerateForeignVocSystem(voc,size) = RecGenerateForeignVocSystem(
  NewForeignVocSystem(voc),size,voc) .

eq GetForeignHead(NewForeignVocSystem(voc)) = ErrorNoWordLeft .
eq GetForeignHead(AddEntryToForeignVocSystem(w,foreignvoc)) = w .
eq GetForeignTail(NewForeignVocSystem(voc)) = ErrorNoForeignEntriesLeft .
eq GetForeignTail(AddEntryToForeignVocSystem(w,foreignvoc)) = foreignvoc .

eq MaxForeignScore(NewForeignVocSystem(voc)) = 0 .
eq MaxForeignScore(AddEntryToForeignVocSystem(w,foreignvoc)) =
  MaxForeignScore(foreignvoc) + Length(
    TranslateIntoNative(GetString(w),GetVocSystem(foreignvoc))) .

eq GetMaxForeign(NewForeignVocSystem(voc),output) = output .
ceq GetMaxForeign(AddEntryToForeignVocSystem(w,NewForeignVocSystem(voc)),
                  output) = w
  if output == DummyValue .
ceq GetMaxForeign(AddEntryToForeignVocSystem(w,NewForeignVocSystem(voc)),
                  output) = output
  if not output == DummyValue .
ceq GetMaxForeign(foreignvoc,output) =
    GetMaxForeign(
  AddEntryToForeignVocSystem(GetForeignHead(foreignvoc),
        GetForeignTail(GetForeignTail(foreignvoc))),
  GetForeignHead(foreignvoc))
  if not GetForeignHead(foreignvoc) == ErrorNoWordLeft
     and not GetForeignTail(GetForeignTail(foreignvoc)) ==
             ErrorNoEntriesLeft
     and not IsStringSmaller(GetString(GetForeignHead(foreignvoc)),
             GetString(GetForeignHead(GetForeignTail(foreignvoc)))) .
ceq GetMaxForeign(foreignvoc,output) =
    GetMaxForeign(GetForeignTail(foreignvoc),GetForeignHead(
                GetForeignTail(foreignvoc)))
  if not GetForeignHead(foreignvoc) == ErrorNoWordLeft
     and not GetForeignTail(GetForeignTail(foreignvoc))
             == ErrorNoEntriesLeft
     and IsStringSmaller(GetString(GetForeignHead(foreignvoc)),
             GetString(GetForeignHead(GetForeignTail(foreignvoc)))) .
ceq GetOrderedForeignWordList(foreignvoc,forout) = forout
  if GetForeignHead(foreignvoc) == ErrorNoWordLeft .
ceq GetOrderedForeignWordList(foreignvoc,forout) =
    GetOrderedForeignWordList(DeleteForeignEntry(
  GetMaxForeign(foreignvoc,DummyValue),foreignvoc),
  AddEntryToForeignVocSystem(GetMaxForeign(foreignvoc,DummyValue),forout))
```

```
      if not GetForeignHead(foreignvoc) == ErrorNoWordLeft .

    eq GetForeignTranslationList(NewForeignVocSystem(voc),lst) = lst .
    eq GetForeignTranslationList(
        AddEntryToForeignVocSystem(w,foreignvoc),lst) =
         GetForeignTranslationList(foreignvoc,
                AddToItemListsList(
                 TranslateIntoNative(GetString(w),
                    GetVocSystem(foreignvoc)),
                 lst)) .
    eq GetForeignOrderedItemList(foreignvoc,lst) =
         GetForeignTranslationList(
           GetOrderedForeignWordList(foreignvoc,
            NewForeignVocSystem(GetVocSystem(foreignvoc))),
           lst) .

    ** The second vocabulary system is a copy of the original one and keeps all
    ** data that is lost from the first one as the first vocsystem is emptied
    ** in order to build the list of all native words .
    eq GetListOfForeignWords(Dictionary(l1,l2,liEmpty),voc) =
         NewForeignVocSystem(voc) .
    eq GetListOfForeignWords(Dictionary(l1,l2,NewItem(it,itlst)),voc) =
         AddToForeignVoc(
           CorrectWord(
       GetForeign(GetPair(it))),GetListOfForeignWords(
                                Dictionary(l1,l2,itlst),voc)) .

    }
}

** ---------------------------------------------------
** Specification of the Test Suite (internal part):
** ---------------------------------------------------

module! TestVocSystemSpec
{
  imports {
    protecting (PreTestVocSystemSpec)
    protecting (ListOfStringListSpec)
    protecting (ListOfPairListsSpec)
    }

  signature {
    [ testnativevocsystem
      testforeignvocsystem
    ]
```

```
  ** ----------------
  ** The Native Part:
  ** ----------------
  op GenerateNativeTest : nativevocsystem Int
                        -> testnativevocsystem { constr }
  op GetNativeWord : testnativevocsystem -> word
  op GetMaxNativeScore : testnativevocsystem -> Nat
  op GetNativeScore : testnativevocsystem -> Int
  op GetNativeVocSystem : testnativevocsystem -> nativevocsystem
  op AdaptNativeVocSystem : String testnativevocsystem -> testnativevocsystem
  op GetNativeStringList : listofitems listofstrings -> listofstrings

  ** -----------------
  ** The Foreign Part:
  ** -----------------
  op GenerateForeignTest : foreignvocsystem Int
                         -> testforeignvocsystem { constr }
  op GetForeignWord : testforeignvocsystem -> word
  op GetMaxForeignScore : testforeignvocsystem -> Nat
  op GetForeignScore : testforeignvocsystem -> Int
  op GetForeignVocSystem : testforeignvocsystem -> foreignvocsystem
  op AdaptForeignVocSystem : String testforeignvocsystem
                           -> testforeignvocsystem
  op GetForeignStringList : listofitems listofstrings  -> listofstrings

  ** -------------------------
  ** Miscellaneous Operations:
  ** -------------------------
  op GetPairList : listofitems pairlist -> pairlist
  op GetListOfPairLists : listoflistofitems listofpairlists
                        -> listofpairlists
  }

axioms {
  var tnvoc : testnativevocsystem
  var tfvoc : testforeignvocsystem
  var nvoc  : nativevocsystem
  var fvoc  : foreignvocsystem
  var pos   : Nat
  var score : Int
  var str   : String
  var vit   : vocitem
  var ilst  : listofitems
  var illst : listoflistofitems
  var sllst : listofpairlists
  var slst  : listofstrings
  var plst  : pairlist
```

```
** ----------------
** The Native Part:
** ----------------
eq GetNativeVocSystem(GenerateNativeTest(nvoc,score)) = nvoc .
eq GetMaxNativeScore(GenerateNativeTest(nvoc,score)) =
    MaxNativeScore(nvoc) .
eq GetNativeScore(GenerateNativeTest(nvoc,score)) = score .
eq GetNativeWord(tnvoc) = GetNativeHead(GetNativeVocSystem(tnvoc)) .
ceq AdaptNativeVocSystem(str,tnvoc) =
     GenerateNativeTest(GetNativeVocSystem(tnvoc),
      GetNativeScore(tnvoc) - 1)
  if MakeWord(str,GetForeignLanguage(GetVocSystem(
                                 GetNativeVocSystem(tnvoc))))
     == ErrorMisspelled .
ceq AdaptNativeVocSystem(str,tnvoc) =
     GenerateNativeTest(GetNativeVocSystem(tnvoc),
      GetNativeScore(tnvoc) + 1)
  if not MakeWord(str,GetForeignLanguage(GetVocSystem(
                                     GetNativeVocSystem(tnvoc))))
     == ErrorMisspelled
  and MakeItem(
       MakePair(
        GetNativeWord(tnvoc),
        MakeWord(str,
           GetForeignLanguage(GetVocSystem(GetNativeVocSystem(tnvoc))))),
       false) in GetItemList(GetVocSystem(GetNativeVocSystem(tnvoc))) .
ceq AdaptNativeVocSystem(str,tnvoc) =
     GenerateNativeTest(GetNativeVocSystem(tnvoc),
      GetNativeScore(tnvoc) - 1)
  if not MakeWord(str,GetForeignLanguage(GetVocSystem(
                                     GetNativeVocSystem(tnvoc))))
     == ErrorMisspelled
  and not MakeItem(
       MakePair(
        GetNativeWord(tnvoc),
        MakeWord(str,
           GetForeignLanguage(GetVocSystem(GetNativeVocSystem(tnvoc))))),
       false) in GetItemList(GetVocSystem(GetNativeVocSystem(tnvoc))) .

eq GetNativeStringList(liEmpty,slst) = slst .
eq GetNativeStringList(NewItem(vit,ilst),slst) =
    GetNativeStringList(ilst,
            AddString(GetNative(GetPair(vit)),slst)) .

** -----------------
** The Foreign Part:
```

```
** ----------------
eq GetForeignVocSystem(GenerateForeignTest(fvoc,score)) = fvoc .
eq GetMaxForeignScore(GenerateForeignTest(fvoc,score)) =
     MaxForeignScore(fvoc) .
eq GetForeignScore(GenerateForeignTest(fvoc,score)) = score .
eq GetForeignWord(tfvoc) = GetForeignHead(GetForeignVocSystem(tfvoc)) .
ceq AdaptForeignVocSystem(str,tfvoc) =
      GenerateForeignTest(GetForeignVocSystem(tfvoc),
      GetForeignScore(tfvoc) - 1)
  if MakeWord(str,GetNativeLanguage(GetVocSystem(
                                 GetForeignVocSystem(tfvoc))))
     == ErrorMisspelled .
ceq AdaptForeignVocSystem(str,tfvoc) =
     GenerateForeignTest(GetForeignVocSystem(tfvoc),
      GetForeignScore(tfvoc) + 1)
  if not MakeWord(str,GetNativeLanguage(GetVocSystem(
                                    GetForeignVocSystem(tfvoc))))
     == ErrorMisspelled
  and MakeItem(
       MakePair(
        MakeWord(str,
           GetNativeLanguage(GetVocSystem(GetForeignVocSystem(tfvoc)))),
        GetForeignWord(tfvoc)),
       false) in GetItemList(GetVocSystem(GetForeignVocSystem(tfvoc))) .
ceq AdaptForeignVocSystem(str,tfvoc) =
      GenerateForeignTest(GetForeignVocSystem(tfvoc),
      GetForeignScore(tfvoc) - 1)
  if not MakeWord(str,GetNativeLanguage(GetVocSystem(
                                    GetForeignVocSystem(tfvoc))))
     == ErrorMisspelled
  and not MakeItem(
       MakePair(
        MakeWord(str,
           GetNativeLanguage(GetVocSystem(GetForeignVocSystem(tfvoc)))),
        GetForeignWord(tfvoc)),
       false) in GetItemList(GetVocSystem(GetForeignVocSystem(tfvoc))) .

eq GetForeignStringList(liEmpty,slst) = slst .
eq GetForeignStringList(NewItem(vit,ilst),slst) =
    GetForeignStringList(ilst,
             NewString(GetForeign(GetPair(vit)),slst)) .

eq GetPairList(liEmpty,plst) = plst .
eq GetPairList(NewItem(vit,ilst),plst) =
    GetPairList(ilst,NewPair(GetPair(vit),plst)) .

eq GetListOfPairLists(lliEmpty,sllst) = sllst .
```

```
    eq GetListOfPairLists(NewItemListsList(ilst,illst),sllst) =
        GetListOfPairLists(illst,NewPairList(
                                  GetPairList(ilst,plEmpty),sllst)) .
    }
}

** -------------------------------
**      The Test Suite:
**      User Interface.
** ------------------------------

module! StudentDictionarySpec
{
imports  {
  protecting ( TestVocSystemSpec )
  }

signature {
  ** ----------------------------
  ** The Score is recorded nicely:
  ** ----------------------------
  [ Score ]
  op _OutOf_ : Int Nat -> Score { constr }

  ** -------------------------------------------
  ** User Interface for modifying the vocabulary:
  ** -------------------------------------------
  op GenerateDictionary : language language -> vocsystem
  op AddEntry : String String vocsystem -> vocsystem
  op DeleteEntry : String String vocsystem -> vocsystem

  ** ---------------------
  ** Translation Utilities:
  ** ---------------------
  op GiveNativeTranslation : String vocsystem -> listofstrings
  op GiveForeignTranslation : String vocsystem -> listofstrings

  ** -----------------
  ** Display Utilities:
  ** -----------------
  op GetAllTranslationsByNativeOrder : vocsystem -> listofpairlists
  op GetAllTranslationsByForeignOrder : vocsystem -> listofpairlists

  ** ------------------
  ** Selftest Utilities:
  ** ------------------
  op TestMyNativeKnowledge : Nat vocsystem -> testnativevocsystem
```

```
  op TestMyForeignKnowledge : Nat vocsystem -> testforeignvocsystem

  op GiveNativeAnswers : listofstrings testforeignvocsystem
                       -> testforeignvocsystem
  op GiveForeignAnswers : listofstrings testnativevocsystem
                        -> testnativevocsystem

  op ShowMyScoreInNative : testnativevocsystem testnativevocsystem -> Score
  op ShowMyScoreInForeign : testforeignvocsystem testforeignvocsystem -> Score
  }

axioms {
  vars l1 l2 : language
  vars s1 s2 : String
  var  lst   : listofstrings
  var  sdic  : vocsystem
  var  no    : Nat
  var  tnvoc : testnativevocsystem
  var  nansw : testnativevocsystem
  var  tfvoc : testforeignvocsystem
  var  fansw : testforeignvocsystem

  eq GenerateDictionary(l1,l2) = Dictionary(l1,l2,liEmpty) .
  eq AddEntry(s1,s2,sdic) = AddEntryToDictionary(s1,s2,sdic) .
  eq DeleteEntry(s1,s2,sdic) = DeleteEntryFromDictionary(s1,s2,sdic) .

  eq GiveNativeTranslation(s1,sdic) =
      GetNativeStringList(TranslateIntoNative(s1,sdic),lsEmpty) .
  eq GiveForeignTranslation(s1,sdic) =
      GetForeignStringList(TranslateIntoForeign(s1,sdic),lsEmpty) .

  eq GetAllTranslationsByNativeOrder(sdic) =
          GetListOfPairLists(
           GetNativeOrderedItemList(
            GetListOfNativeWords(sdic,sdic),
       lliEmpty),lplEmpty) .
  eq GetAllTranslationsByForeignOrder(sdic) =
     GetListOfPairLists(
      GetForeignOrderedItemList(
       GetListOfForeignWords(sdic,sdic),
       lliEmpty),lplEmpty) .

  eq TestMyNativeKnowledge(no,sdic) =
      GenerateNativeTest(GenerateNativeVocSystem(sdic,no),0) .
  eq TestMyForeignKnowledge(no,sdic) =
      GenerateForeignTest(GenerateForeignVocSystem(sdic,no),0) .
```

```
  eq GiveNativeAnswers(lsEmpty,tfvoc) =
      GenerateForeignTest(
       DeleteForeignEntry(
              GetForeignHead(
               GetForeignVocSystem(tfvoc)),
         GetForeignVocSystem(tfvoc)),
             GetForeignScore(tfvoc)) .
  eq GiveNativeAnswers(NewString(s1,lst),tfvoc) = GiveNativeAnswers(
        lst,AdaptForeignVocSystem(s1,tfvoc)) .

  eq GiveForeignAnswers(lsEmpty,tnvoc) =
       GenerateNativeTest(
             DeleteNativeEntry(
              GetNativeHead(
               GetNativeVocSystem(tnvoc)),
              GetNativeVocSystem(tnvoc)),
             GetNativeScore(tnvoc)) .
  eq GiveForeignAnswers(NewString(s1,lst),tnvoc) = GiveForeignAnswers(
      lst,AdaptNativeVocSystem(s1,tnvoc)) .

  eq ShowMyScoreInNative(nansw,tnvoc) =
       GetNativeScore(nansw) OutOf GetMaxNativeScore(tnvoc) .
  eq ShowMyScoreInForeign(fansw,tfvoc) =
       GetForeignScore(fansw) OutOf GetMaxForeignScore(tfvoc) .
  }
}
```

3.2 An English-to-German Student Dictionary

In this section, we will demonstrate the implementation of the student dictionary system we created in the previous chapter. We will construct a dictionary between the English and the German language and demonstrate the user interface of our cafeOBJ implementation. We have used indentation to clarify the cafeOBJ input and output to some extend.
Let us build a simple English-German dictionary.

```
%StudentDictionarySpec> op testDict : -> vocsystem .

%StudentDictionarySpec> eq testDict = AddEntry("car","Auto",
  AddEntry("car","Wagen",
    AddEntry("frog","Frosch",
      AddEntry("paper","Papier",
        AddEntry("table","Tisch",
          AddEntry("blackboard", "Tafel",
            AddEntry("table","Tabelle",
              GenerateDictionary(English,German))))))).
```

Let us see what it is like.

```
%StudentDictionarySpec> reduce testDict .
* -- reduce in % : testDict
Dictionary(
  MakeLanguage(
    NewRule("qx",NewRule("uu",rEmpty)),
    NewChar('A',NewChar('a',NewChar('B',NewChar('b',NewChar('C',NewChar('c',
      NewChar('D',NewChar('d',NewChar('E',NewChar('e',NewChar('F',NewChar(
      'f',NewChar('G',NewChar('g',NewChar('H',NewChar('h',NewChar('I',
      NewChar('i',NewChar('J',NewChar('j',NewChar('K',NewChar('k',NewChar(
      'L',NewChar('l',NewChar('M',NewChar('m',NewChar('N',NewChar('n',
      NewChar('O',NewChar('o',NewChar('P',NewChar('p',NewChar('Q',NewChar(
      'q',NewChar('R',NewChar('r',NewChar('S',NewChar('s',NewChar('T',
      NewChar('t',NewChar('U',NewChar('u',NewChar('V',NewChar('v',NewChar(
      'W',NewChar('w',NewChar('X',NewChar('x',NewChar('Y',NewChar('y',
      NewChar('Z',NewChar('z',
        coEmpty))))))))))))))))))))))))))))))))
    ))))))))))))))))))))),
  MakeLanguage(
    NewRule("yy",NewRule("jj",rEmpty)),
    NewChar('A',NewChar('a',NewChar('B',NewChar('b',NewChar('C',NewChar(
      'c',NewChar('D',NewChar('d',NewChar('E',NewChar('e',NewChar('F',
      NewChar('f',NewChar('G',NewChar('g',NewChar('H',NewChar('h',NewChar(
      'I',NewChar('i',NewChar('J',NewChar('j',NewChar('K',NewChar('k',
      NewChar('L',NewChar('l',NewChar('M',NewChar('m',NewChar('N',NewChar(
      'n',NewChar('O',NewChar('o',NewChar('P',NewChar('p',NewChar('Q',
      NewChar('q',NewChar('R',NewChar('r',NewChar('S',NewChar('s',NewChar(
      'T',NewChar('t',NewChar('U',NewChar('u',NewChar('V',NewChar('v',
     NewChar('W',NewChar('w',NewChar('X',NewChar('x',NewChar('Y',NewChar(
     'y',NewChar('Z',NewChar('z',coEmpty))))))))))))))))))))))))))))
    ))))))))))))))))))))))))),
  NewItem(MakeItem(
    MakePair(CorrectWord("car"),
             CorrectWord("Auto")),
    false),
  NewItem(MakeItem(
    MakePair(CorrectWord("car"),
             CorrectWord("Wagen")),
    false),
  NewItem(MakeItem(
    MakePair(CorrectWord("frog"),
             CorrectWord("Frosch")),
    false),
  NewItem(MakeItem(
    MakePair(CorrectWord("paper"),
             CorrectWord("Papier")),
```

```
    false),
  NewItem(MakeItem(
    MakePair(CorrectWord("table"),
             CorrectWord("Tisch")),
    false),
  NewItem(MakeItem(
    MakePair(CorrectWord("blackboard"),
             CorrectWord("Tafel")),
    false),
  NewItem(MakeItem(
    MakePair(CorrectWord("table"),
             CorrectWord("Tabelle")),
    false),
  liEmpty)))))))) : vocsystem

(0.000 sec for parse, 47165 rewrites(8.930 sec), 61237 match attempts)
```

Let us use it to translate a couple of words in both directions.

`%StudentDictionarySpec > reduce GiveNativeTranslation( "Wagen",testDict) .`

– reduce in NewString("car",lsEmpty) : listofstrings (0.000 sec for parse, 47367 rewrites(9.570 sec), 61549 match attempts)

`%StudentDictionarySpec > reduce GiveForeignTranslation( "car",testDict) .`

– reduce in NewString("Auto",NewString("Wagen",lsEmpty)) : listofstrings (0.000 sec for parse, 47351 rewrites(9.610 sec), 61523 match attempts)

Let us build a test for testing the knowledge of English words.

`%StudentDictionarySpec > op nativeKnowledgeTest : -> testnativevocsystem .`

`%StudentDictionarySpec > eq nativeKnowledgeTest = TestMyNativeKnowledge(4,testDict) .`

Let us look at the test.

`%StudentDictionarySpec > reduce nativeKnowledgeTest .`

– reduce in
GenerateNativeTest(AddEntryToNativeVocSystem(CorrectWord("blackboard") ,AddEntryToNativeVocSystem(CorrectWord("frog"),AddEntryToNativeVocSystem(CorrectWord("table"),AddEntryToNativeVocSystem(CorrectWord("car") ,NewNativeVocSystem(Dictionary(MakeLanguage(NewRule("qx",NewRule("uu",rEmpty)),NewChar('A',NewChar('a',NewChar('B',NewChar('b',NewChar('C',NewChar('c',NewChar('D',NewChar('d',NewChar('E',NewChar('e', NewChar('F',NewChar('f',NewChar('G',NewChar('g',NewChar('H',NewChar('h',NewChar('I',NewChar('i',NewChar('J',NewChar('j',NewChar('K', NewChar('k',NewChar('L',NewChar('l',NewChar('M',NewChar('m',NewChar(

'N',NewChar('n',NewChar('O',NewChar('o',NewChar('P',NewChar('p',
NewChar('Q',NewChar('q',NewChar('R',NewChar('r',NewChar('S',NewChar(
's',NewChar('T',NewChar('t',NewChar('U',NewChar('u',NewChar('V',
NewChar('v',NewChar('W',NewChar('w',NewChar('X',NewChar('x',NewChar(
'Y',NewChar('y',NewChar('Z',NewChar('z',coEmpty))))))))))))))))
))))))))))))))))))))))))))))))))))))),MakeLanguage(NewRule("yy",
NewRule("jj",rEmpty)),NewChar('A',NewChar('a',NewChar('B',NewChar(
'b',NewChar('C',NewChar('c',NewChar('D',NewChar('d',NewChar('E',
NewChar('e',NewChar('F',NewChar('f',NewChar('G',NewChar('g',NewChar(
'H',NewChar('h',NewChar('I',NewChar('i',NewChar('J',NewChar('j',
NewChar('K',NewChar('k',NewChar('L',NewChar('l',NewChar('M',NewChar(
'm',NewChar('N',NewChar('n',NewChar('O',NewChar('o',NewChar('P',
NewChar('p',NewChar('Q',NewChar('q',NewChar('R',NewChar('r',NewChar(
'S',NewChar('s',NewChar('T',NewChar('t',NewChar('U',NewChar('u',
NewChar('V',NewChar('v',NewChar('W',NewChar('w',NewChar('X',NewChar(
'x',NewChar('Y',NewChar('y',NewChar('Z',NewChar('z',coEmpty))))
))),NewItem(MakeItem(
MakePair(CorrectWord("car"),CorrectWord("Auto")),false),NewItem(
MakeItem(MakePair(CorrectWord("car"),CorrectWord("Wagen")),false)
,NewItem(MakeItem(MakePair(CorrectWord("frog"),CorrectWord("Frosch")
),false),NewItem(MakeItem(MakePair(CorrectWord("paper"),CorrectWord(
"Papier")),false),NewItem(MakeItem(MakePair(CorrectWord("table")
,CorrectWord("Tisch")),false),NewItem(MakeItem(MakePair(CorrectWord(
"blackboard"),CorrectWord("Tafel")),false),NewItem(MakeItem(MakePair(
CorrectWord("table"),CorrectWord("Tabelle")),false),liEmpty))))
)))))))))),0) : testnativevocsystem
(0.000 sec for parse, 49076 rewrites(10.470 sec), 64000 match attempts)

Let us translate the first entry, i.e. blackboard into German

```
%StudentDictionarySpec > reduce GiveForeignAnswers(
AddString("Tafel",lsEmpty),nativeKnowledgeTest) .
```

– reduce in
GenerateNativeTest(AddEntryToNativeVocSystem(CorrectWord("frog"),
AddEntryToNativeVocSystem(CorrectWord("table"),AddEntryToNativeVocSystem(
CorrectWord("car"),NewNativeVocSystem(Dictionary(MakeLanguage(NewRule(
"qx",NewRule("uu",rEmpty)),NewChar('A',NewChar('a',NewChar('B',
NewChar('b',NewChar('C',NewChar('c',NewChar('D',NewChar('d',NewChar(
'E',NewChar('e',NewChar('F',NewChar('f',NewChar('G',NewChar('g',
NewChar('H',NewChar('h',NewChar('I',NewChar('i',NewChar('J',NewChar(
'j',NewChar('K',NewChar('k',NewChar('L',NewChar('l',NewChar('M',
NewChar('m',NewChar('N',NewChar('n',NewChar('O',NewChar('o',NewChar(
'P',NewChar('p',NewChar('Q',NewChar('q',NewChar('R',NewChar('r',
NewChar('S',NewChar('s',NewChar('T',NewChar('t',NewChar('U',NewChar(
'u',NewChar('V',NewChar('v',NewChar('W',NewChar('w',NewChar('X',
NewChar('x',NewChar('Y',NewChar('y',NewChar('Z',NewChar('z',coEmpty)
)),MakeLanguage(

NewRule("yy",NewRule("jj",rEmpty)),NewChar('A',NewChar('a',NewChar(
'B',NewChar('b',NewChar('C',NewChar('c',NewChar('D',NewChar('d',
NewChar('E',NewChar('e',NewChar('F',NewChar('f',NewChar('G',NewChar(
'g',NewChar('H',NewChar('h',NewChar('I',NewChar('i',NewChar('J',
NewChar('j',NewChar('K',NewChar('k',NewChar('L',NewChar('l',NewChar(
'M',NewChar('m',NewChar('N',NewChar('n',NewChar('O',NewChar('o',
NewChar('P',NewChar('p',NewChar('Q',NewChar('q',NewChar('R',NewChar(
'r',NewChar('S',NewChar('s',NewChar('T',NewChar('t',NewChar('U',
NewChar('u',NewChar('V',NewChar('v',NewChar('W',NewChar('w',NewChar(
'X',NewChar('x',NewChar('Y',NewChar('y',NewChar('Z',NewChar('z',
coEmpty))),NewItem(
MakeItem(MakePair(CorrectWord("car"),CorrectWord("Auto")),false)
,NewItem(MakeItem(MakePair(CorrectWord("car"),CorrectWord("Wagen")
),false),NewItem(MakeItem(MakePair(CorrectWord("frog"),CorrectWord(
"Frosch")),false),NewItem(MakeItem(MakePair(CorrectWord("paper")
,CorrectWord("Papier")),false),NewItem(MakeItem(MakePair(CorrectWord(
"table"),CorrectWord("Tisch")),false),NewItem(MakeItem(MakePair(
CorrectWord("blackboard"),CorrectWord("Tafel")),false),NewItem(
MakeItem(MakePair(CorrectWord("table"),CorrectWord("Tabelle")),
false),liEmpty)))))))))))),1) : testnativevocsystem
(0.000 sec for parse, 51769 rewrites(10.790 sec), 67501 match attempts)

We can see our score.

```
%StudentDictionarySpec > reduce ShowMyScoreInNative(
GiveForeignAnswers(AddString("Tafel",
AddString("Wrong",lsEmpty)),nativeKnowledgeTest),
nativeKnowledgeTest) .
```

– reduce in "Tafel",AddString("Wrong",lsEmpty)),nativeKnowledgeTest),nativeKnowledgeTest)
1 OutOf 6 : Score (0.000 sec for parse, 108253 rewrites(22.250 sec), 141389 match attempts)

We can see all the translations contained in the dictionary ordered by foreign order.

```
%StudentDictionarySpec > reduce GetAllTranslationsByForeignOrder(
testDict) .
```

– reduce in
NewPairList(NewPair(MakePair(CorrectWord("car"),CorrectWord("Auto")
),plEmpty),NewPairList(NewPair(MakePair(CorrectWord("frog"),CorrectWord(
"Frosch")),plEmpty),NewPairList(NewPair(MakePair(CorrectWord("paper")
,CorrectWord("Papier")),plEmpty),NewPairList(NewPair(MakePair(CorrectWord(
"table"),CorrectWord("Tabelle")),plEmpty),NewPairList(NewPair(MakePair(
CorrectWord("blackboard"),CorrectWord("Tafel")),plEmpty),NewPairList(
NewPair(MakePair(CorrectWord("table"),CorrectWord("Tisch")),plEmpty)
,NewPairList(NewPair(MakePair(CorrectWord("car"),CorrectWord("Wagen")
),plEmpty),lplEmpty))))))) : listofpairlists
(0.000 sec for parse, 50669 rewrites(10.870 sec), 67258 match attempts)

Chapter 4

Conclusions

Let us end this paper with some conclusions we drew during and after the specification of the data type of our *Student Dictionary System.*

We think it has become clear how big the formal specification of a simple data type can become, if one chooses to build the specification up from the very basics that are available (as we did). This became clear to us at the point when we started alternating writing the specification with implementing parts of the specification in the cafeOBJ language.

We tried to avoid this phenomenon by defining tools that would allow us to re-use specifications by extending them, but even then the specification became rather large. For even more speed-up, we used specifications already implemented in cafeOBJ (like string and list) instead of implementing these ourselves. The speed gained by the use of these existing cafeOBJ packages, however, was again lost by some missing tools in cafeOBJ: we especially missed the availability of if-then-else statements, forcing us to explicitly write each possibility in a separate if-statement. The other major tool we lacked in cafeOBJ was the notion of local definition blocks of the form

let x = ... in ...

This also had an impact on some parts of the written formal specification, which were more or less designed simultaneous with their cafeOBJ counterparts. In order to save time, we opted to keep the differences between the cafeOBJ implementation and the written formal specification minimal, meaning we did not use the notion of local definition blocks everywhere in the written specification.

This lead us to conclude that formal specification of an entire data type quite often is too time-consuming to be profitable. Rather, one should concentrate on the difficult or interesting parts of the data type and formalize the specification of this part. This not only saves time but also allows the designer to get insight in the important levels of the data type rather that to lose himself in the lower, supporting levels of the data type.

Bibliography

[G$^+$93] Joseph A. Goguen et al. Introducing obj. Technical report, Programming Research Group, Oxford University, UK, October 1993.

[GM85] Narain Gehani and Andrew McGettrick. *Software Specification Techniques.* Addison-Wesley, Wokingham, England, 1985.

[LEW96] Jacques Loeckx, Hans-Dieter Ehrich, and Markus Wolf. *Specification of Abstract Data Types.* Wiley & Teubner, Chichester, UK, 1996.

[NSF98] Ataru T. Nakagawa, Toshimi Sawada, and Kokichi Futatsugi. *CafeOBJ User's Manual – ver. 1.4*, 1998. Available from http://theta.theta.ro/cafeobj.

[PST96] Ben Potter, Jane Sinclair, and David Till. *An introduction to Formal Specification and Z.* International Series in Computer Science. Prentice Hall, London, UK, second edition, 1996.

[vHL89] Ivo van Horebeek and Johan Lewi. *Algebraic Specifications in Software Engineering.* Springer, Berlin, Germany, 1989.