Gefördert vom
bmb+f
Bundesministerium für
Bildung, Wissenschaft,
Forschung und Technologie

Verbmobil
Verbundvorhaben

A Library Package for the Verbmobil Interface Term

Michael Dorna

Universität Stuttgart

Vm

Report 238
May 2000

May 2000

Michael Dorna

Institut für Maschinelle Sprachverarbeitung
Universität Stuttgart
Azenbergstraße 12
D – 70174 Stuttgart

Tel.: (0711) 121 - 1363
Fax: (0711) 121 - 1366

Gehört zum Antragsabschnitt: 3 Übersetzung

Die vorliegende Arbeit wurde im Rahmen des Verbundvorhabens Verbmobil vom Bundesministerium für Bildung, Wissenschaft, Forschung und Technologie (BMBF) unter dem Förderkennzeichen 01 IV 701 N3 gefördert. Die Verantwortung für den Inhalt dieser Arbeit liegt bei dem Autor.

ISSN 1434-8845

Abstract

This report describes the Prolog implementation of an ADT package for the "Verb*mobil* Interface Term" (VIT). After a short presentation of the contents of a VIT the functionality of the software is presented. In several components of the Verb*mobil* systemthe library package is used, e.g., for creating, for manipulating, for printing and for checking the contents of a VIT. This report is an updated version of Verb*mobil* report 104.

The edition of this documentation corresponds to version 2.10 of the ADT package. An on-line HTML documentation is available via WWW at `http://www.ims.uni-stuttgart.de/verbmobil/vitADTX`.

Contents

1 Introduction

1.1 The VIT Library Package

The Verb*mobil* Interface Term (VIT) is one of the still existing "inventions" of Verb*mobil*-1. The same holds for the ADT library package which handles it. No more needs to be said. Well, this report is an update of Verb*mobil* report 104 [Dorna (1996)] where you may find the first part of the story.

1.2 The Verb*mobil* Interface Term

The VIT is used as a uniform data structure at the interfaces between several software components of the Verb*mobil* system. It is an encoding of different linguistically motivated pieces of information produced and used in these components.

The contents of a VIT correspond to a segmented utterance in a dialogue turn. This partitioning of turns enables the linguistic components to work incrementally.

The main contents of a VIT are semantic representations. On the other hand, information like morpho-syntax, syntactic tense, semantic sorts, scope, prosody, and the analyzed surface string is also part of a VIT. This information is linked to semantics and can be used for computing semantic tense, for disambiguation of underspecified analyses, for guiding semantic evaluation such as anaphora resolution, for adjacency or linear precedence determination, and for many more.

In a large project like Verb*mobil* data abstraction is an important basis for the parallel development of different components which should communicate with each other in the end. Hence, also from a software engineering perspective there are a lot arguments for the VIT. Among them there are:

- Because there are abstract access and manipulation operations available the data structures of the VIT can be changed with minor feedback on a software component using it.
- We assume that this is the first time that the result of linguistic components is checked using a kind of analysis structure control. Language-specific on-line dictionaries are used to insure the compatibility between components. A content checker is used to test structural properties and a "deep" checker checks even semantically relevant wellformedness. It has been shown that this treatment is well-suited for error detection in components with a rapidly growing linguistic coverage. Furthermore, the complex information produced by linguistic components even make automatic output control necessary.
- The checkers can be used to define a quality rating, e.g. for correctness, interpretability, etc. of the contents of a VIT. Such results are much better

to improve a system than common, purely quantitative, measures based on failure or success rates.

- The single VIT representation serves as a common "language" (*lingua franca*) for discussions. This language can be "spoken" by people responsible for technical tasks only as well as others with purely linguistic intentions.

1.3 Multiple Levels of Information

The contents of a VIT are filled into the following slots:

Slot Name	Description
VIT ID	combines a unique tag for a turn segment described by the current VIT and the word lattice path used in its linguistic analysis;
Index	a triple consisting of the entry points for traversing the VIT representation;
Conditions	labelled conditions describing the possibly underspecified semantic content of an utterance;
Constraints	scope and grouping constraints, e.g. used for underspecified quantifier and operator scope representation;
Sorts	sortal specifications for instance variables introduced in labelled conditions;
Discourse	additional semantic and pragmatic information, e.g. discourse roles for individual instances;
Syntax	morpho-syntactic features, e.g. number and gender of individual instances;
Tense and Aspect	morpho-syntactic tense combined with aspect and sentence mood information, e.g. used for computing surface tense;
Prosody	prosodic information such as accenting and sentence mood.

Beside "VIT ID" and "Index" all slots are encoded as variable free lists of terms. The lists are used because they are very easy to manipulate. In typical AI languages such as Lisp and Prolog they are built-in and they can be ported easily to other programming languages. In general, the list elements do not introduce any further recursive embedding, i.e. the elements are like fixed records with fields containing constants. The minimally recursive representation was chosen for efficient information access.

The information within the several slots is connected to information in other slots by identical constants which can be seen as bidirectional links. These constants – we call them *holes*, *labels* and *instances* – could be interpreted as skolemized logical variables which denote a node in a graph. I.e., a sharing like this combines information of different nature or source which, therefore, is part of different slots in a VIT.

1.4 Contents of VIT Slots

This section describes the information which can be found in the slots of a VIT. First, we list the terms,[1] and then we explain the possible argument bindings.

1.4.1 VIT ID

The "VIT ID" slot is filled with a VIT identifier of the form

```
vitID(SegmentID, WHGString)
```

`SegmentID` is a unique marker for each VIT. The Prolog encoding looks like

```
sid(TurnNo, Channel, SourceLanguage, BeginTime, EndTime,
    Reading, CurrentLanguage, TurnEnd, Sender)
```

E.g. `sid(104,a,ge,7002,7199,1,jp,y,transfer)` is an example of a segment identifier with a string representation `t104ageb7002e7199r1jpytransfer`.[2]

`WHGString` is a Prolog list which contains edges of a word lattice. This sequence of edges describes the surface string selected by a syntactic/semantic component for producing the current VIT. An edge can have either one of the following forms:

```
word(String, WID, ListOfLs)
filler(String, WID)
```

`String` is the label of the edge and `WID` is a unique edge identifier. Both, `String` and `WID`, are Prolog atoms. If `String` is a linguistic information, the `word/3` representation is used. For more technical information such as pauses, noice, etc. `filler/2` is used.

The list of labels `ListOfLs` links the edges to some labelled conditions in the Conditions slot. If there is more than one label in a single list, a word was decomposed into several lexemes in semantic analysis. If there are identical labels in different lists, the corresponding words form one labelled condition together. Hence, this encoding reflects an n-to-m mapping between syntax and semantics during linguistic analysis.

1.4.2 Index

The "Index" slot is filled with a term of the form

```
index(TopHole,MainLabel,MainInst)
```

where the hole `TopHole`, the label `MainLabel`, and the instance `MainInst` are the entry points for traversing a VIT.

---

[1] We describe the Prolog representations only. There exists others, e.g. in LISP.

[2] See [Alexandersson et al. (1997a), Auerswald (1997)] for more information on segment identifiers used in Verb*mobil* and the value ranges of their fields. Note that channels `s` and `t` are reserved for string and treebank input, repectively.

1.4.3 Conditions

The contents of the "Conditions" slot are language specific.[3] In general, it contains terms of the form

```
Functor(Label, Arg2, ..., Argx)
```

called *labelled conditions*. The semantic entities are, e.g., predicates, roles, operators, and quantifiers. The first argument – sometimes called *base label* – is always a unique identifier for such an object. The semantic variables for labels and markers, such as events, states and individuals, are skolemized with special constant symbols, e.g. `l1` for a label and `i1` for a state.

The labeling of semantic conditions is very useful since the recursive embedding of argument structure, operator scope, etc. is no longer syntactically represented in a recursive data structure but achieved through the interpretation of additional labeling constraints. In this respect, label arguments act as pointers to the corresponding arguments. Additionally, all these special constants can be seen as pointers for adding or linking information within and between multiple slots of the VIT.

The set of all possible conditions for one language is defined by its lexicon. The ADT package supports on-line lexicons for arbitrary languages. See Appendix D for the expected format and for hooks to extend a language specific lexicon. Examples of such lexicons can be found in `vitSemLex.pl` for English, German, and Japanese.

[3] See [Kasper et al. (2000), **?**] for the concrete semantic representation classes used in Verb*mobil*. For information on the language specific semantic databases, see [Flickinger (2000), Kasper (2000), Shimada (2000)].

.

1.4.4 Other Slots

The information located in the rest of the slots is given in the following table:

Info	Description	Slot Name
ana_ante(Inst,ListOfIs)	anaphoric antecedents	Discourse
c_class(Label,CClass)	coordination type	Discourse
cas(Inst, Case)	syntactic case	Syntax
dialog_act(Label, DialogAct)	dialogue act information	Discourse
dialog_phase(DialogPhase)	dialogue phase	Discourse
disc_honor(Inst,Inst)	honorifics (for Japanese)	Discourse
disc_role(Inst,DiscRole)	discourse role	Discourse
disc_stat(Label,Status)	discourse status	Discourse
discourse_function(L,DiscFunc)	pragmatic discourse relation	Sorts
dir(Label, YesNo)	(non)directional preposition	Discourse
eq(Label,LabelOrHole)	equality constraint	Constraints
gend(Inst, Gender)	morpho-syntactic gender	Syntax
in_g(Label, GroupLabel)	label is in group	Constraints
leq(Label,Hole)	scope/subordination constraint	Constraints
nom_ante(Inst,ListOfIs)	anaphoric reference	Discourse
num(Inst, Number)	morpho-syntactic number	Syntax
pcase(Label,Inst,Atom)	subcategorized preposition	Syntax
pers(Inst, Person)	person	Syntax
prontype(I, PRef, PType)	type of a pronoun	Discourse
pros_accent(Label)	prosodic accent	Prosody
pros_boundary(Label)	prosodic (b3) marker	Prosody
pros_emph(Label)	emphasis marking	Prosody
pros_mood(Label, PMood)	prosodic mood	Prosody
pros_probab(Label, Prob)	prosodic accent probability	Prosody
ref_ante(Inst, Inst)	anaphora resolution	Discourse
rel_ante(Inst,ListOfIs)	anaphoric reference	Discourse
s_class(Label, SClass)	disambiguation class	Sorts
s_sort(Inst, Sort)	sortal restriction	Sorts
sem_focus(Label, LabelOrHole)	focus resolution	Constraints
subj_honor(Inst,Inst,Honor)	subject honorific marker (Japanese)	Syntax
syn_cat(Label,Atom)	syntactic category	Syntax
ta_aspect(Inst, Aspect)	aspectual information	Tense and Aspect
ta_mood(Inst, TMood)	mood	Tense and Aspect
ta_perf(Inst, Perf)	perfect	Tense and Aspect
ta_tense(Inst, Tense)	surface tense	Tense and Aspect
topic(Label,GroupLabel,Topic)	topic information	Discourse
unbound(LabelOrInstOrHole)	unbound markers	Discourse

The argument values of the terms above and of the predicates described in the next sections are given in the following table:[4]

[4] For the sort and dialogue act values and their relations, i.e. the semi-lattices, see Section 7.5 and [Alexandersson et al. (1998)], respectively.

Argument	Description or list of values
Arg$x$	Hole, Inst, Label, Atom, or a Prolog list of the named Args
ArgType	np, acomp, comp, xcomp
Aspect	progr, nonprogr plus prospective, terminative for Japanese only
Atom	Prolog atomic
BeginTime	unsigned integer
Case	;(Case,Case), nom, gen, dat, acc, noc(ase)
Check	ground, shape
Channel	a, b, c, d, e, f, g, s, t
Class	ambig, disc, grad, mod, mood, noun, quant, verb
CurrentLanguage	see Language
CClass	m, n, q, s, s
DialogAct	;(DialogAct,DialogAct), ~(DialogAct), accept, backchannel, bye, clarify, close, commit, confirm, control_dialogue, defer, deliberate, deviate_scenario, dialogue_act, digress, exclude, explained_reject, feedback, feedback_negative, feedback_positive, give_reason, greet, inform, init, introduce, manage_task, offer, politeness_formula, promote_task, refer_to_setting, reject, request, request_clarify, request_comment, request_commit, request_suggest, signal_non_understanding, suggest, thank
DialogPhase	hello, opening, negotation, closing, good_bye
DiscFunc	attitude, check, coherence, edit, emphasize, exemplify, hesitate, indifferent,, known, negative, nonpragmatic, pop, positive, prefer, push, repair, revised, smoothen, structure, surprise, uptake
DiscRole	he, sp
EndTime	unsigned integer
Functor	Prolog functor
Gender	;(Gender,Gender), fem, masc, neut
GramFunc	subj, obj, obj2, nof(unction)
Hole or H	Prolog atom of the form h[a-z][0-9]*[5]
Honor	minus, plus
Info	any valid term encoding information in a VIT
Inst or I	Prolog atom of the form i[a-z][0-9]*
Label or L	Prolog atom of the form l[a-z][0-9]*
Language	de, en, ge, jp
ListOfLs	Prolog list of Label elements
ListOfIs	Prolog list of Inst elements
TMood	ind, conj, imp
Number	;(Number,Number), sg, pl
Perf	perf, nonperf
Person	;(Person,Person), 1, 2, 3
PRef	sp, he, sp_he, third, third_he, top
Prob	an integer i with $0 \leq i \leq 100$

... *continued on next page*

Argument	Description or list of values
PType	refl, std, refl_std, recip, imp, event, event_std, demon, demon_event, zero (and intersent for Japanese)
PMood	decl, prog, quest
Reading	integer
SemRole	arg1, arg2, arg3, refl
Sender	Prolog atom which is the name of a Verb*mobil* component
String	Prolog atom encoding a LaTeX string
SClass	sa, fp, mp, dra
SMood	decl, imp, quest, decl_quest, decl_imp, imp_quest, decl_imp_quest
SlotName	'Conditions', 'Constraints', 'Tense and Aspect', 'Index', 'Prosody', 'Discourse', 'Sorts', 'VIT ID', 'Syntax'
Sort	;(Sort,Sort) or &(Sort,Sort) or ~(Sort) or abstract, action_sit, agentive, animal, anything, communicat_sit, entity, field, food, geo_location, human, info_bearer, info_content, institution, instrument, location, meeting_sit, mental_sit, move_sit, nongeo_location, object, position_sit, property, situation, space_time, substance, symbol, temp_sit, temporal, thing, time, vehicle
SourceLanguage	see Language
Status	old, new, inferred_old, contrast_new
Stream	a valid Prolog stream object
Tense	pres, past, future
Topic	fr, mo, vf, wa
TurnEnd	n, y
TurnNr	unsigned integer
VIT	Verb*mobil* Interface Term (not necessarily ground)
YesNo	yes, no, yesno

1.5 The Prolog Implementation

In Prolog, the VIT is implemented as a term of arity 9 named `vit`. The name and arity of this realization should be of no interest if the access to a VIT is always handled by the ADT package (or a similar abstraction). If this is the case, we are not restricted to this implementation in future. As already pointed out, in general, the information is encoded in lists and the elements are terms. This data structure is very flexible in adding and removing information, i.e. the terms.

The output of each component dealing with VITs has to make sure the following properties:[6]

- A VIT is (logical) variable free, i.e. all variables are skolemized.

[5] The letter after the i encodes the producer of the instance symbol (c: chunk parser; d: dialogue summary; h: HPSG parser; k: context evaluation; r: robust semantics; s: statistical parser; t: transfer). For offline processing this information is optional. All this holds for label and hole symbols, too.

[6] Of course, all partners are free to use any other representations, e.g. lexicon extensions (see Appendix D), *within* their components.

- A VIT contains only valid information, i.e. in general, lists of terms with atomic arguments.
- The syntax of a term in a VIT is unique with respect to the slot it belongs to. I.e. the functor and arity are sufficient to uniquely determine the adequate slot. Additionally, for labelled conditions the functor and arity are sufficient to uniquely determine the semantic class it belongs to.
- There is only a fixed number of possible terms allowed in a VIT which are restricted to a fixed number of possible argument values (enumerations) or to specific types (e.g., integers).

For describing the predicates of the ADT package we use the "standard" notation for call patterns (aka mode information):

`+` the argument is expected to be instantiated (not necessarily ground) and will not be changed during processing of the called predicate;

`-` the argument is expected to be a variable and will be bound during processing of the called predicate;

`?` the argument can be instantiated when calling the predicate and/or will be bound during processing of the called predicate.

The ADT package is realized as a Prolog module named `vitADT` which exports the predicates described in the following sections. For further remarks on the usage see Appendix B.

1.6 Overview

The rest of this documentation is organized as follows. In Section 2 we present the predicates for constructing a new VIT and filling it with information. In Section 3 and Section 4 we outline predicates for information access and those for deleting information, respectively. Section 5 informs about predicates for checking the contents of a VIT. In Section 6 we show predicates for printing a VIT. Miscellaneous predicates are described in Section 7.

Appendix A explains how to get and install the ADT package and Appendix B shows how to use it. Appendix C introduces the term conversion package `atom2term` which comes with the library package distribution. Finally, Appendix D sketches briefly the contents and possible extensions of the on-line dictionaries contained in `vitSemLex.pl` and `vitValues.pl`.

For each predicate presented in this document we give the call pattern(s) and a brief description sometimes including an example.

2 Create and Fill a VIT

2.1 Create a new VIT

`vitNew(-VIT)`

Generates a new VIT which can be filled, e.g. using `vitAdd/2`. At calltime `VIT` should be a variable.

Example:

```
| ?- vitNew(VIT).

VIT = vit(_099,_100,_101,_102,_103,_104,_105,_106,_107)
```

2.2 Add Information to a VIT

In the current implementation, adding of information to a new generated VIT is handled by open ended lists. The correct location for the information to be added is handled by the syntax, i.e. by the form of a term the designated slot can be determined automatically. The behaviour of `vitAdd/{2,3}` can be influenced using `vitLazyCheck/0` or `vitRegularCheck/0` (see Section 5.3).

`vitAdd(+Infos, ?VIT)`

Adds information to the contents of a VIT. `vitAdd/2` assumes open ended list arguments for adding information. Hence, the VIT should not be closed (see Section 2.3 below). `Infos` can be a single `Info` or a list of `Info` elements. See Section 7.5 for automatic conversion of `Infos` to VIT internal representations.

Example:

```
| ?- vitNew(VIT),
     vitAdd(s_sort(i1,thing),VIT).

VIT = vit(_88,_89,_90,_91,[s_sort(i1,thing)|_105],_92,_93,_94,_95)
```

`vitAdd(+Info, +VIT, ?VIT)`

Once the open ended lists of a VIT are closed `vitAdd/3` should be used to add new information to a VIT. It adds a single information `Info` to the contents of a VIT computing a new or modified VIT. See Section 7.5 for automatic conversion of `Infos` to VIT internal representations.

Example:

```
| ?- vitNew(V),
     vitAdd([stat(l7,i3),equal(l7,i6,i8)],V),
     vitClose(V),
     vitAdd(s_sort(i1,thing),V,V1).

V = vit(_1,_2,[stat(l7,i3),equal(l7,i6,i8)],[],[],[],[],[],[]),
V1 = vit(_1,_2,[stat(l7,i3),equal(l7,i6,i8)],[],[s_sort(i1,thing)],[],[],[],[])
```

2.3 Close a VIT

`vitClose(?VIT)`

Closes open ended lists in arguments of a VIT, if necessary.

Example:

```
| ?- vitNew(VIT),
     vitAdd(s_sort(i1,thing),VIT),
     vitClose(VIT).

VIT = vit(_48,_50,[],[],[s_sort(i1,thing)],[],[],[],[])
```

2.4 Manipulate the "VIT ID" Slot

`vitNewSpan(+BeginTime,+EndTime,+VIT,-VIT)`

Replaces the time span values of a given VIT.

`vitNewWHGString(+WHGString,+VIT,-VIT)`

Replaces the WHGString value of a given VIT.

`vitNewLanguage(+CurrentLanguage,+VIT,-VIT)`

Replaces the CurrentLanguage value of a given VIT.

`vitNewSender(+Sender,+VIT,-VIT)`

Replaces the Sender value of the current VIT.

3 Access to Information of a VIT

In this section all predicates are listed which can be used to access the contents of a VIT. We have divided these into two groups: the ones which *access the slots* of a VIT and the ones which give *direct access to the Prolog terms* within a VIT.

3.1 Accessing the Slots

The slot access predicates have the form `vit*(?VIT,?SlotContent)` and can be used to manipulate a whole slot. For valid information see Section 1.4.

`vitConditions(?VIT, ?Conditions)`

Unifies `Conditions` with the Conditions slot of a VIT.

`vitConstraints(?VIT, ?Constraints)`

Unifies `Constraints` with the Constraints slot of a VIT.

`vitDiscourse(?VIT, ?Discourse)`

Unifies `Discourse` with the Discourse slot of a VIT.

`vitID(?VIT, ?VITID)`

Unifies `VITID` with the "VIT ID" slot.

`vitIndex(?VIT, ?Index)`

Unifies `Index` with the Index slot of a VIT.

`vitProsody(?VIT, ?Prosody)`

Unifies `Prosody` with the Prosody slot of a VIT.

`vitSorts(?VIT, ?Sorts)`

Unifies `Sorts` with the Sorts slot of a VIT.

`vitSyntax(?VIT, ?Syntax)`

Unifies `Syntax` with the Syntax slot of a VIT.

`vitTenseAspect(?VIT, ?TenseAspect)`

Unifies `TenseAspect` with the "Tense and Aspect" slot of a VIT.

3.2 Accessing Specific Information

The information access predicates have the form `vit*(+InstOrLabel,+VIT,?Value)` taking an instance or label and a VIT resulting a value if the information is available in the VIT. In general, the following predicates fail if the required information is not available. For value ranges see Section 1.4.

3.2.1 Accessing Morpho-syntactic Information

`vitAspect(+Inst, +VIT, ?Aspect)`

Reports the aspect value for a given instance.

`vitCase(+Inst, +VIT, ?Case)`

Reports the case value for a given instance. See Section 7.5 for VIT internal representations of morpho-syntactic values.

`vitMood(+Inst, +VIT, ?TMood)`

Reports the mood value for a given instance (of a verb).

`vitPerf(+Inst,+VIT,?Perf)`

Reports the perfect value for a given instance (of a verb).

`vitPerson(+Inst, +VIT, ?Person)`

Reports the person value for a given instance. See Section 7.5 for VIT internal representations of morpho-syntactic values.

`vitSentMood(+VIT, ?SMood)`

Reports the sentence mood for a given `VIT`. `SMood` is the functor of the condition with label `MainLabel` (restricted to the sentence mood conditions).

`vitSynGender(+Inst, +VIT, ?Gender)`

Reports the gender value for a given instance. See Section 7.5 for VIT internal representations of morpho-syntactic values.

`vitSynNumber(+Inst, +VIT, ?Number)`

Reports the number for a given instance. See Section 7.5 for VIT internal representations of morpho-syntactic values.

`vitTense(+Inst, +VIT, ?Tense)`

Reports the tense value for a given instance.

3.2.2 Accessing Status Infomation

Note: The following predicates can be used to instantiate the specific arguments of the segment id term `sid/9` or the index term `index/3` of a given VIT, respectively.

`vitChannel(?VIT,?Channel)`

Reports the input channel of the original data.

`vitID(?VIT,?SegmentID,?WHGString)`

Gives access to the `vitID/2` information.

`vitIndex(?VIT,?TopHole,?MainLabel,?MainInst)`

Gives access to the `index/3` information.

`vitLanguage(?VIT,?CurrentLanguage)`

Reports the current language of the VIT.

`vitReading(?VIT,?Reading)`

Reports the number of the current analysis (reading) of an input.

`vitSender(?VIT,?Sender)`

Reports the producer module of the VIT.

`vitSourceLanguage(?VIT,?SourceLanguage)`

Reports the language of the original input data.

`vitSpan(?VIT,?BeginTime,?EndTime)`

Reports the time span covered by the VIT.

`vitTurn(?VIT,?TurnNr)`

Reports the turn number of the VIT.

`vitTurnEnd(?VIT,?TurnEnd)`

Reports the turn-end flag value.

3.2.3 Accessing Sortal and other Classifications

`vitClass(+Label,+VIT,-Class)`

Reports the disambiguation class for a given label of an underspecified condition.

`vitSort(+Inst, +VIT, ?Sort)`

Reports the sort value for a given instance. See Section 7.5 for VIT internal sort representations.

`vitDir(+Label, +VIT, ?YesNo)`

Given the label of a preposition this predicate looks if it is marked directional (`YesNo = yes`) or non-directional (`YesNo = no`).

`vitDiscourseFunction(+Label, +VIT, -DiscFunc)`

Reports a discourse function value.

`vitPCase(+/?Label, ?/+Inst, +VIT, -Atom)`

Given a label or an instance the pcase value will be reported.

3.2.4 Accessing Dialogue Information

`vitDialogAct(+VIT, +Label, ?DialogAct)`

Reports the dialogue act associated with the main label `Label` of a VIT. See Section 7.5 for VIT internal dialogue act representations.

`vitDialogPhase(+VIT, ?DialogPhase)`

Reports the dialogue phase of a VIT.

`vitPronType(+Inst, +VIT, ?PRef, ?PType)`

Reports the reference and type of a pronoun with instance `Inst`.

3.2.5 Accessing Scopal Constraints

`vitEqual(?Label,+VIT,?Hole)`

Checks if a label and a hole are marked as equal, i.e. the VIT contains an `eq(Label, Hole)` term (backtrackable).

`vitInGroup(?Label,+VIT,?GroupLabel)`

Looks for a single grouping (backtrackable).

`vitSubOrd(?Label,+VIT,?Hole)`

Unifies with a subordination information `leq(Label, Hole)` (backtrackable).

3.2.6 Accessing Prosodic Information

`vitProsAccent(+Label, +VIT)`

Checks if there is a prosodic accent mark on the condition with label `Label`.

`vitProsBound(+Label, +VIT)`

Checks if there is a prosodic boundary on the condition with label `Label`.

`vitProsMood(+Label, +VIT, ?PMood)`

Reports the prosodic mood value of the condition with label `Label`.

3.2.7 Miscellaneous

`vitAmbig(+VIT, -Ambiguities)`

Reports all labelled conditions in the Conditions slot which belong to one of the lexical ambiguity classes (see [Kasper et al. (2000)]). `Ambiguities` is a list of such conditions or `[]` if there are none.

4 Copy and Delete Information

4.1 Copy VITs and Selected Slots

`vitCopy(+VIT, -VIT)`

Makes a copy of a VIT which shares no variables with its original.

Example:

```
| ?- VIT = vit(A,[],B,[s_sort(i1,thing)],[],[],[],[],[],[]),
     vitCopy(VIT,NewVIT).

VIT = vit(A,[],B,[s_sort(i1,thing)],[],[],[],[],[],[]),
NewVIT = vit(_8290,[],_8292,[s_sort(i1,thing)],[],[],[],[],[],[])
```

`vitCopyAllBut(+Integer, ?VIT, ?VIT)`

Makes a copy of all slots of a VIT without copying argument `Integer`. This predicate should always be used together with `vitSlotName/2` (see Section 7.7).

Example:

```
| ?- VIT = vit(A,[],B,[s_sort(i1,thing)],[],[],[],[],[],[]),
     vitSlotName('Sorts',No),
     vitCopyAllBut(No,VIT,NewVIT).

VIT = vit(A,[],B,[s_sort(i1,thing)],[],[],[],[],[],[]),
NewVIT = vit(A,[],B,_8300,[],[],[],[],[],[])
```

4.2 Delete Information of a VIT

`vitDelete(?Info, +VIT, ?VIT)`

Deletes information from a given VIT if it exists. Otherwise `vitDelete/3` simply unifies the output with the given VIT. `Info` cannot be a variable itself when calling `vitDelete/3` but can contain some. I.e. `vitDelete/3` can be used for further instantiating `Info` (not backtrackable).

Example:

```
| ?- vitDelete(s_sort(i1,X),
               vit(u1,[],l1,[s_sort(i1,thing)],[],[],[],[],[],[]),
               VIT).

X = thing,
VIT = vit(u1,[],l1,[],[],[],[],[],[],[])
```

5 Check a VIT or Parts of It

This section lists all predicates which can be used to check the VIT or parts of it. The basic tests check the syntactical wellformedness wrt. valid information described in Section 1.4 and the language specific lexicons. Further tests run on the contextual relations between pieces of information in a VIT (see Section 5.1).

After error detection several options regulate the reporting and behaviour of the ADT package. Other options control the language selection and the treatment of lexical gaps in the on-line lexicons (see Section 5.3).

Special predicates check the wellformedness of single pieces of information, its accordance to slots and the type of its arguments (see Section 5.4).

5.1 VIT Checkers

`vitCheck(+VIT)`

Combines `vitCheckFormat/1` and `vitCheckContent/1` (see below).

`vitCheckFormat(+VIT)`

Checks the syntax of the information in a given VIT. Currently, the checking covers the following:

- the compatibility with the semantic lexicon using the current language of the VIT; e.g. argument frames for verbs, argument value ranges of `Info` terms, etc.;
- existence of type information for pronouns;
- existence of directionality information for prepositions.

`vitCheckContent(+VIT)`

Checks dependencies between information in a given VIT. Currently, the checking covers the following:

- missing predicates for labels found in `WHGString`;
- existence of minimal morpho-syntactic tense for verb instances;
- unique `Info` assignments to instances and labels in the slots `'Discourse'`, `'Prosody'`, `'Sorts'`, `'Syntax'`, and `'Tense and Aspect'`;
- unique prosodic assignments to labels;
- existence of an `leq/2` or `eq/2` entry for a hole;
- detection of cyclic groupings;

- missing groupings for group labels in Conditions or Constraints;
- unexpected groupings (for non-group labels);

`vitDeepCheck(+VIT)`

Checks connections between information in a given VIT.[7] Currently, the checking covers the following:

- proper groupings, e.g., a label is not both a base label and a group label, or a group is empty or redundant;
- proper `leq/2` and `eq/2`, e.g., an `eq/2` is not a `les/2` (see [Kasper et al. (2000)]) or is not redundant;
- single top label or top hole, i.e. the graph encoded in a VIT is not multi-rooted;
- instances should be bound uniquely;
- cycles, i.e. the label and hole relations given by `leq/2`, `eq/2` and `les/2` do not build a cycle.

`vitCheckGroupings(+VIT,-VIT)`

Checks and repairs groupings, e.g., removes nested groupings or tries to break cyclic groupings.

`vitCheckSubord(+VIT,-VIT)`

Checks and repairs `eq/2` constraints, top labels, cycles, and redundant `leq/2` constraints.

5.2 Error Reporting Options

The error reporting can be toggled by `vitReportErrors/0` (default) and `vitDontReportErrors/0`. The error handler's output can be redirected using `vitErrorOutput/1` and reset to the default output (`user_error`) by `vitResetErrorOutput/0`. After errors have been detected and reported any further processing can be influenced using `vitIgnoreError/0` (default) or `vitFailOnError/0`.

`vitReportErrors`

All detected errors are reported after checking (default).

`vitDontReportErrors`

No errors are reported even if some were detected during checking.

[7] The original version of this checker was developed by Michael Schiehlen.

`vitIgnoreError`

Switches the checkers error handler to "ignore", i.e. even if an error occured, the checkers succeed (default).

`vitFailOnError`

Switches the checkers error handler to "fail", i.e. an error forces a failure.

`vitErrorOutput(+File)`

Redirects the checkers error output to a file or stream. `File` needs to be an accessible and writeable file or a Prolog output stream. Default is `user_error`.

`vitResetErrorOutput`

Switches error output to the default output, i.e. sets output to `user_error`.

5.3 Lexicon Compatibility Options

The current language used by the ADT package can be switched using `vitSetLanguage/1`. `vitSetMyLanguage/1` is used to switch to private on-line lexicons. The current language setting can be accessed using `vitLanguage/1`. `vitAutoLanguage/0` uses the current language specified in the "VIT ID" slot for checking. With `vitIgnoreLanguage/0` the checkers ignore the language of a VIT and use the current (manual) language setting.

`vitLazyCheck/0` changes the behaviour of `vitCheckFormat/1` and `vitAdd/{2,3}` to be lazy with respect to the language specific parts of a VIT. This is important when using information which is not known to the ADT package. This kind of information is assumed to be part of the Conditions slot. The system will report every detection of unknown information during processing but will not fail. To switch off this mode use `vitRegularCheck/0` (default).

`vitAutoLanguage`

The ADT selects a VIT's language, e.g. for checking the contents, based on its `CurrentLanguage` value (default).

`vitIgnoreLanguage`

The current ADT language has to be manually specified, e.g. using `vitSetLanguage/1`.

`vitLanguage(?Language)`

Reports current language setting.

`vitSetLanguage(+Language/+VIT)`

Switches the ADT to a given language or the current language of a VIT (encoded in the SegmentID).

`vitSetMyLanguage(+Atom)`

Switches the ADT to a user defined lexicon in a Prolog module named `Atom`.

`vitLazyCheck`

Using this mode `vitCheckFormat/1` and `vitAdd/{2,3}` try to handle syntactically unknown information.

`vitRegularCheck`

This mode assumes all information in a VIT to be known to the ADT package (default).

5.4 Checking Information Terms

Single information can be checked using `vitValidInfo/{2,3}` or `vitADT:validInfoCheck/3`. The different types of arguments can be tested with `vitInst/1`, `vitLabel/1`, `vitHole/1`, and `vitLabelOrHole/1`.

`vitValidInfo(+Info, +Check)`

Same as `vitValidInfo(Info, Check, _)` (see below).

`vitValidInfo(+Info, +Check, ?Slot)`

Checks syntax of a given information and if it is valid for a slot. If `Check == shape`, only the form (functor and arity) of `Info` will be checked. In any other case, a regular check will be performed assuming `Info` to be a valid instance.

Example:

```
| ?- vitValidInfo(s_sort(i1,_),shape,S).

S = 'Sorts'

| ?- vitValidInfo(s_sort(i1,thing),ground,'Semantics').

no
| ?- vitValidInfo(s_sort(i1,thing),ground,Slot).

Slot = 'Sorts'
```

`vitADT:validInfoCheck(+Info,?Slot,-Code)`

Reports checks for a given information `Info`. The valid slot for `Info` is unified with `Slot`. `Code` is callable Prolog code for checking the arguments of an information.[8] Note: This predicate is **not** exported by the `vitADT` module!

Example:

```
| ?- vitADT:validInfoCheck(pers(X,Y),Slot,Code).

Slot = 'Syntax',
Code = vitInst(X),personValue(Y)
```

5.5 Tests on Argument Types

There are type checks implemented for all kinds of information in a VIT. If you need some other tests than the ones described below, check the code!

`vitInst(+Inst)`

Checks the syntax of an instance.

Example:

```
| ?- vitInst(l1).

no
| ?- vitInst(it1).

yes
```

`vitLabel(+Label)`

Checks the syntax of a label.

`vitHole(+Hole)`

Checks the syntax of a hole.

`vitLabelOrHole(+LabelOrHole)`

Checks the syntax of either a label or a hole.

[8] For hackers: all Prolog predicates used in `Code` are defined in either `vitBasicTypes.pl` or `vitValues.pl` and can be seen in the on-line lexicon `vitSemLex.pl` (see Appendix D).

6 Print a VIT

`vitPrint/{1,2}` pretty-print a given VIT such that it can be used as a Prolog term again. `vitPrintRaw/{1,2}` prints a VIT without any formating. `vitPrintSpecial/{1,2}` prints a VIT formated in groupings and collects information for instances and labels.

`vitPrint(+VIT)`

Pretty-prints a VIT (using current output stream).

Example:

```
vit( vitID(sid(2,a,de,0,10,1,de,y,synger),          % SegmentID
           [word(das,1,[128]),                       % WHG String
            word(ist,2,[126]),
            word('wundersch"on',3,[125])]),
     index(124,125,i7),                              % Index
     [stat(125,i7),                                  % Conditions
      wunderschoen(125,i8),
      decl(124,h4),
      pron(128,i8)],
     [leq(123,h4),                                   % Constraints
      in_g(126,123),
      in_g(128,123),
      in_g(125,134)],
     [],                                             % Sorts
     [prontype(i8,third,demon)],                     % Discourse
     [gend(i8,neut),                                 % Syntax
      num(i8,sg),
      pers(i8,3),
      cas(i8,nom)],
     [ta_mood(i7,ind),                               % Tense and Aspect
      ta_perf(i7,nonperf),
      ta_tense(i7,pres)],
     [pros_mood(124,decl)]                           % Prosody
   )
```

`vitPrint(+Stream,+VIT)`

Pretty-prints a given VIT like `vitPrint/1` directing the output to a given stream.

`vitPrintRaw(+VIT)`

Prints a given VIT without comments, linebreaks, etc. and without any further conversions. See Section 7.5 for not printing VIT internal representations.

`vitPrintRaw(+Stream,+VIT)`

Prints a given VIT like `vitPrintRaw/1` directing the output to a given stream.

vitPrintSpecial(+VIT)

Pretty-prints a given VIT and checks for non-referring labels or holes and superfluous groupings.[9] The output is enclosed in Prolog block comment markers.

The output consists of three parts: The first line shows the string which is covered by the VIT. The next two parts are grouped conditions and additional information for labels and instances. There are three columns in the latter parts: label or instance, VIT information, and further information. If labels are not used elsewhere in a VIT, they are not printed and you see a _ instead.

In the conditions part horizontal lines are used to separate groupings. Group labels and subordination information are printed to the right of each line. If conditions are not grouped, a question mark ? appears in the output. Below the conditions part additional information available for instances and labels is printed. In general, attribute-value pairs are listed there.

Example:

```
/*
das ist wundersch"on

---------------------------------- l23 main label, =< h4
_       [support(i7,l34)]              : ist
_       [pron(i8)]                     : das
---------------------------------- l34
_       [wunderschoen(i8)]             : wunderschoen
------------------------------------ ?
l24     [decl(h4)]                     top label

i8      [cas        : nom,
         pers       : 3,
         num        : sg,
         gend       : neut,
         prontype   : third+demon]
i7      [ta_tense   : pres,
         ta_perf    : nonperf,
         ta_mood    : ind]
l24     [pros_mood  : decl]

no errors detected
*/
```

vitPrintSpecial(+Stream,+VIT)

Pretty-prints a given VIT like `vitPrintSpecial/1` directing the output to a given stream.

9 The original version of this pretty-printer was developed by Michael Schiehlen.

7 Miscellaneous Operations on VITs

7.1 Manipulate Semantic Conditions

`vitLabelCond(+LCond/-LCond, ?Label, -Cond/+Cond)`

Decomposes a given labelled condition `LCond` into a label `Label` and a condition `Cond` or constructs `LCond` of `Label` and given `Cond`.

Example:

```
| ?- vitLabelCond(support(l2,i1,l5),L,C).

L = l2,
C = support(i1,l5)

| ?- vitLabelCond(LC,L,support(i1,l5)).

LC = support(L,i1,l5)
```

`vitPredicate(+Condition,?Label,?Inst)`

Succeeds if `Condition` is of VIT class `verb`, has base label `Label` and instance `Inst`.

`vitExistsPred(?Label,?Inst,+VIT,-Condition)`

Succeeds if the given VIT contains a labelled condition `Condition` for which `vitPredicate(Condition,Label,Inst)` holds. If `Label` or `Inst` are already instantiated, `vitExistsPred/4` is deterministic. Otherwise it is backtrackable, i.e. can be used for look-up.

`vitExistsCond(+Class,+VIT,?ListOfArgs)`

Matches the list `ListOfArgs` with the arguments of a condition which is of class `Class` (for values of `Class` see Section 1.4). The arity of the anonymous condition has to be greater or equal to the length of `ListOfArgs`. The arguments are matched from left to right.

`vitCondition(?[Label,Arg2|Args],?Class,+VIT,?Cond)`

Looks for a condition `Cond` with base label `Label`, second argument `Arg2` and rest of arguments `Args` in `VIT` (backtrackable). The search can be restricted by a given class `Class`. Note: No roles like `arg`X`/3` will be matched!

7.2 Collect Information about an Instance

`vitInstInfo(+Inst,+VIT,-Infos)`

For a given VIT, `vitInstInfo/3` collects all information about an instance in a list of `Attribute:Value` pairs.

7.3 Traversing a VIT

`vitAccessibleLabels(+Label/?Label,?Label/+Label,+VIT)`

Checks whether two labels are connected by a transitive closure over labelled conditions and subordination constraints (backtrackable). Given a label `Label1/Label2` this predicates enumerates all labels `Label2/Label1` for which `Label2` is subordinated by `Label1`. This closure is restricted by scopal islands given by modal verbs as well as sentential complements.

`vitTopHole(+QuaOrScop,+BottomLabel,+VIT,-TopHole)`

Given the name of a quantifier or scopal modifier `QuaOrScop` and a (group) label `BottomLabel`, vitTopHole/4 returns a specific top hole `TopHole` when moving up the VIT graph starting at `BottomLabel`.

`TopHole` can be used to include new constituents which should have an embedding level dependent on the kind of `QuaOrScop`. Sentence moods, modals, subordinations and `arg3`s with hole arguments are used as barriers during seaching. For `whq`s and scopal modifiers barriers are considered, for all kind of quantifiers these were ignored.

Instead of using specific quantifiers or operators one can use the key words `global_scope`, `local_scope`, or `operator_scope`. Using `global_scope`, the value returned for `TopHole` will be the topmost hole in the VIT wrt. the starting point, even if the VIT is not wellformed and it has multiple top holes. Using `local_scope` `vitTopHole/4` returns the hole of the next barrier, and `operator_scope` returns the first hole found, possibly of an operator.

7.4 Operations on the Surface String

The following predicates operate on the `WHGString` of a VIT.

`vitAdjacent(+Label1,+Label2,+VIT)`

Checks if the given labels belong to adjacent words in the WHG string of the VIT. I.e. the base labels belong to edges which occur adjacent in the list of edges.

`vitLinearPrecedence(+Label1,+Label2,+VIT)`

Checks if the given labels belong to words which are in linear precedence order in the WHG string of the VIT. Note that the computation ignores auxilliaries.

`vitFullForm(+Label,+VIT,-ListOfWords)`

Returns the list of words in the `WHGString` of a VIT which are associated with a given label. If the list contains more than one element, the order corresponds to the surface order of the words. If the label does not correspond to any word, the list is empty.

`vitSentenceInitial(+Label,+VIT)`

Checks if the given label belongs to the first word in the WHG string.

`vitSentenceFinal(+Label,+VIT)`

Checks if the given label belongs to the last word in the WHG string.

`vitString(+VIT,-String)`

Concatenates the string values of the edges which form the WHG string.

7.5 Computing with Types

7.5.1 Background

The sort values used in VITs are part of a sort hierarchy, called *United Sorts* (see Figure 1), and the dialogue acts are also part of a dialogue act hierarchy.[10] These hierarchies are used to compare individual values, e.g. with respect to their compatibility or coverage.

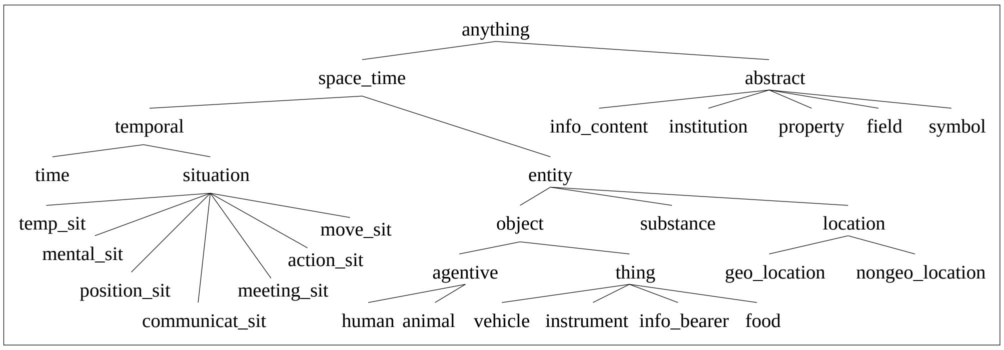

Figure 1: "United Sorts of Verb*mobil*"

The ADT package uses a special bit vector encoding of these latices. The morpho-syntactic information within a VIT, which can consist of disjunctive values, is converted to bit vectors, too.[11] The VIT internal representations are computed when a VIT is filled using `vitAdd/`{`2,3`}. We call these representations *types* and an instance thereof is a `Type`.

In general, the internal encoding is invisible but it is included in the result of the following operations:

10 Note that `anything` is not a valid sort value within a VIT, i.e. the checker will raise an error if the value of a sort is equivalent to `anything`. The same holds for the most general dialogue act value, called `dialog_act`. Hence, in both cases, this information should be left out if no more specific values can be inferred locally.

11 See `vitHierarchy.pl` for the code.

- printing a VIT using `vitPrintRaw/{1,2}`;
- accessing sortal information using `vitSort/3`;
- accessing dialogue acts using `vitDialogAct/3`;
- accessing morpho-syntactic information using `vitCase/3`, `vitPerson/3`, `vitSynGender/3`, `vitSynNumber/3`.

7.5.2 Conversion of Internal Representations

In some cases it is necessary to convert a VIT with internal sort and dialogue act encoding into a VIT with the equivalent symbolic encoding. The same holds for the internal representation of morpho-syntactic information. The following predicates will do this job.

`vitExtern2Intern(+VIT,-VIT)`

Takes a VIT which may contain symbolic sort, dialogue act or morpho-syntactic information and converts it into a VIT with the internal bit vector encoding.

`vitIntern2Extern(+VIT,-VIT)`

Takes a VIT which may contain the internal bit vector encoding for sorts, dialogue acts, or morpho-syntactic information and converts it into a VIT with symbolic encoding only.

7.5.3 Operations on Types

`vitTypeCheck(+Type1,+Type2)`

This operation is used for checking selectional restrictions where `Type1` is the actual value and `Type2` is the expected one. `vitTypeCheck(T1,T2)` holds if either `vitTypeEqual(T1,T2)` or `vitTypeUnify(T1,T2,T12)` and not `vitTypeEqual(T12,T2)`.

`vitTypeConvert(+Extern/-Extern,-Type/+Type)`

Converts an external sort, dialogue act, or morpho-syntactic value into the internal type format or vice versa.

`vitTypeEqual(+Type1,+Type2)`

Checks the (denotational) equality of the given types.

`vitTypeRealSubsumes(+Type1,+Type2)`

`vitTypeRealSubsumes(T1,T2)` holds if `vitTypeSubsumes(T1,T2)` and not `vitTypeSubsumes(T2,T1)`.

`vitTypeSubsumes(+Type1,+Type2)`

Holds if `Type1` is subsumed by `Type2`, i.e. `Type2` is more general than `Type1` or, in other words, `Type2` covers `Type1`.

`vitTypeUnify(+Type1,+Type2,-Type12)`

`Type12` is the intersection (or conjunction) of `Type1` and `Type2`. If there is no intersection, `vitTypeUnify/3` fails.

`vitTypeUnifiable(+Type1,+Type2)`

Same as `vitTypeUnify(Type1,Type2,_)`, i.e. it checks the compatibility of `Type1` and `Type2`.

7.6 ADT Package Status

The ADT package can be initalized using `vitInit/{0,1}` and the version can be checked with `vitVersion/1`. The current ADT package settings are reported by `vitSettings/0`.

`vitInit`

Initializes the ADT package. `vitInit/0` is called automatically when loading the package.

`vitInit(+VITID)`

Initializes the ADT package by calling `vitInit/0` and sets the language of the package using `vitSetLanguage(CurrentLanguage)`, where `CurrentLanguage` is determined by the VIT id.

`vitVersion(?Version)`

Reports the version of the ADT package in use.

`vitSettings`

Reports current settings such as error reporting, error output and language.

Example:

```
| ?- vitSettings.

% VIT-ADT settings:
% vitAutoLanguage
% vitRegularCheck
% vitIgnoreError
% vitReportErrors
% vitErrorOutput(user_error)
% vitLanguage(de)
```

7.7 Miscellaneous

`vitSlotName/2` can be used to list all slot names together with their argument position in the current VIT implementation.

`vitSlotName(?Name,?Integer)`

`Name` is a valid slot name and `Integer` its argument number in the current implementation (backtrackable).

Example:

```
| ?- vitSlotName('Index',I).

I = 2
```

References

[Alexandersson et al. (1997a)] J. Alexandersson, M. Auerswald, M. Dorna, W. Finkler, T. Ruland, C.J. Rupp, and J. Schwinn. *The Art of Naming Segments.* Verb*mobil* Memo 119, DFKI GmbH Saarbrücken, May 1997. Available via WWW `ftp://VM-depot@ftp.dfki.de/FTP-SERVER/vm-memos/memo-119-97.ps.gz` (access restricted to Verb*mobil* partners).

[Alexandersson et al. (1998)] J. Alexandersson, B. Buschbeck-Wolf, T. Fujinami, M. Kipp, S. Koch, E. Maier, N. Reithinger, B. Schmitz, and M. Siegel. *Dialogue Acts in* Verb*mobil-2 – Second Edition.* Verb*mobil* Report 226. DFKI GmbH, Universities of Berlin, Saarbrücken and Stuttgart, July 1998. Available via WWW `http://verbmobil.dfki.de/cgi-bin/verbmobil/htbin/decode.cgi/share/VM-depot/FTP-SERVER/vm-reports/report-226-98.ps.gz`.

[Auerswald (1997)] Marko Auerswald. *Neue Nachrichtenkennung für ICE.* Verb*mobil* Technical Document 60, DFKI GmbH Kaiserslautern, June 1997. Available via WWW `ftp://VM-depot@ftp.dfki.de/FTP-SERVER/vm-techdocs/techdok-60-97.ps.gz` (access restricted to Verb*mobil* partners).

[Dorna (1996)] Michael Dorna. *The ADT Package for the* Verb*mobil Interface Term.* Verb*mobil* Report 104, IMS, Universität Stuttgart, July 1996. Available as PostScript or HTML document via WWW `http://www.ims.uni-stuttgart.de/verbmobil/vitADT/`.

[Flickinger (2000)] Dan Flickinger (ed.) *English Semantic Database.* Version of April 4, 2000. Available via WWW `ftp://VM-depot@ftp.dfki.de/EXCHANGE/semdbs/` (access restricted to Verb*mobil* partners).

[Kasper (2000)] Walter Kasper (ed.). *German Semantic Database.* Version of April 6, 2000. Available via WWW `ftp://VM-depot@ftp.dfki.de/EXCHANGE/semdbs/` (access restricted to Verb*mobil* partners).

[Kasper et al. (2000)] W. Kasper, J. Bos, M. Schiehlen and C. Thielen. *Definition of the Abstract Semantic Classes.* Version of March 10, 2000. Available via WWW `http://www.dfki.de/~kasper/vm/class-doku/class-doku.html` (access restricted to Verb*mobil* partners).

[Modulmanager (1998)] *Modulmanager.* DFKI GmbH, Kaiserslautern, 1998. Available via WWW `http://www.dfki.uni-kl.de/verbmobil/doc/lib/moduleManager/moduleManager.ps` (access restricted to Verb*mobil* partners).

[PCA (1998)] *PCA für Prolog.* DFKI GmbH, Kaiserslautern, 1998. Available via WWW `http://www.dfki.uni-kl.de/`

verbmobil/doc/lib/pools/PCA_Prolog.ps (access restricted to Verb*mobil* partners).

[Quintus Manual (1995)] *Quintus Prolog User's Manual.* Quintus Prolog Release 3.2, Quintus Corporation, Mountain View, California, 1995

[Schiehlen (1999)] M. Schiehlen. *Semantikkonstruktion*, volume 5 of *Arbeitspapiere des IMS (AIMS).* IMS, Universität Stuttgart. Doctorial thesis.

[Shimada (2000)] Atsuko Shimada (ed.). *Japanese Semantic Database.* Version of March 30, 2000. Available via WWW ftp://VM-depot@ftp.dfki.de/EXCHANGE/semdbs/ (access restricted to Verb*mobil* partners).

[SICStus Manual (1995)] *SICStus Prolog User's Manual.* Release 3, Swedish Institute of Computing Science, Kista, Sweden, June 1995.

A Getting and Installing the Software

To run the software you need either Quintus Prolog [Quintus Manual (1995)], at least Version 3.2, or SICStus Prolog [SICStus Manual (1995)], Version 3, at least patchlevel 3. For installation you need GNU make.

The ADT package is available via anonymous ftp from the Verb*mobil* ftp-server `ftp.dfki.uni-sb.de` at DFKI, Saarbrücken (access restricted to Verb*mobil* partners). Then follow these steps:

```
ftp> cd EXCHANGE/VIT/
ftp> bin
ftp> get vitADT_2_10_0.tgz
or
ftp> get vitADT_2_10_0_bin.tgz
```

where the latter is a binary distribution only.

Afterwards you have to unzip and untar the file in your local `$VM_HOME` directory, e.g. using

```
% gtar -zxf vitADT_2_10_0.tgz
```

That's it! Please see the `README` file in `$VM_HOME/doc/lib/Vit_Adt/` for recent changes and there you may find the newest edition of this document, too.

B Usage

B.1 Using the Package

The name of the ADT package is vitADT.{pl,qof,ql}. It is a usual library package which defines a module named vitADT. Hence it can be consulted like every other Prolog file containing a module, e.g. by calling ensure_loaded(vitADT) or use_module(vitADT, *<list of imported predicates>*).

If predicates of the ADT package are not imported into the current module, predicate calls have to be prefixed with the module name vitADT, e.g. like vitADT:vitPrint(VIT). If predicates are imported, the prefix can be omitted; i.e. the same call may look like vitPrint(VIT).

B.2 Error Messages

The package produces error messages if the call patterns are violated. There are two types of errors. Both types cause a predicate to fail. First, there are instatiation errors: an argument was expected to be instatiated at call time. The error looks like

```
! Instatiation error: <something> expected but <this> found
! Goal: vitADT:<goal>
```

The second type of errors are domain errors: an argument was instatiated with a wrong argument type. Then the error message looks like

```
! Domain error: <something> expected but <this> found
! Goal: vitADT:<goal>
```

C Module `atom2term`

C.1 Introduction

The module `atom2term` exports predicates which can be used to convert terms to atoms or strings (lists of character codes) and vice versa. These can be used together with PCA [PCA (1998), Modulmanager (1998)] if a Prolog component wants to use the message type `idl_string` which is identical to a Prolog atomic.

A VIT has to be ground by definition if a component sends or receives one (see Section 1.5). **If a VIT is not variable-free calling `term2atomic/2` or `term2string/2`, `numbervars/3` will be called.** I.e. all variables will be instantiated with `'$VAR'(`*Integer*`)` terms. Note: These `'$VAR'/1` terms might be invisible during debugging and may cause strange errors because they look like regular variables!

C.2 Predicates

`term2atomic(+Term, -Atomic)`

Converts a term `Term` to an atomic `Atomic` if necessary; inversion of `atomic2term/2`.

Example:

```
| ?- term2atomic(term(arg1(a),arg2(b)),A).

A = 'term(arg1(a),arg2(b))'
```

`atomic2term(+Atomic, -Term)`

Converts an atomic `Atomic` to a term `Term` if necessary; inversion of `term2atomic/2`.

Example:

```
| ?- atomic2term('term(arg1(a),arg2(b))',T).

T = term(arg1(a),arg2(b))
```

`term2string(+Term, -String)`

Converts a term `Term` to a list of character codes `String`; inversion of `string2term/2`.

Example:

```
| ?- term2string(term(arg1(a),arg2(b)),A).

A = "term(arg1(a),arg2(b))"
```

`string2term(+String, -Term)`

Converts a list of character codes `String` to a term `Term`; inversion of `term2string/2`.

Example:

```
| ?- string2term("term(arg1(a),arg2(b))",T).

T = term(arg1(a),arg2(b))
```

D On-line Lexicon Databases

D.1 Built-in Semantic Lexicon Database

The ADT package contains the files `vitSemLex.pl`, `vitValues.pl` and `vitLinking.pl` which define on-line lexicon databases and syntax-semantic linking tables. Currently, there exist lexicons and linking information for German based on [Kasper (2000)], for English based on [Flickinger (2000)] and for Japanese based on [Shimada (2000)]. All three are based on the semantic class descriptions of [Kasper et al. (2000)].

The data which is part of the current ADT package was extracted of the named resources using UNIX shell scripts such as `awk`, `sed`, etc. The result is mainly a reduction to relation names and semantic classes. Hence, an on-line lexicon does not contain the whole information of the original database, e.g. decomposition classes are no longer covered. Contextual constraints such as grouping, subordination, etc. are also not part of this database. In general, the values of these constraints are the result of a semantic construction process.

D.2 Format of a On-line Database

An entry in a lexicon database has the following form:

```
Language:vitSemLex(LabelledCondition,
                   SemanticClass,
                   Class,
                   ConditonCheckingCode,
                   ContextOfCondition,
                   ContextCheckingCode,
                   SortalRestrictions) :- Goals.
```

As can be seen, `Language` defines a module in which a lexicon entry `vitSemLex/7` can be accessed.

The first argument is always a labelled condition `LabelledCondition` which should belong to the defined language.

The second argument contains the semantic class `SemanticClass` of the condition. The potential values are described in [Kasper et al. (2000)].

The syntax checking code `ConditonCheckingCode` and `ContextCheckingCode` is used, e.g., by `vitCheckFormat/1` (see `vitCheck.pl`). The callable predicates are either Prolog built-ins, library predicates, defined in `vitValues.pl` or already explained in previous sections of this documentation.

The context list `ContextOfConditon` is given by the class description. Together with parts of the checking code like `vitGroupLabel/1` this information is used in `vitCheckContent/1`. In general, this kind of information is optional information which may occur together with the condition in a VIT.

The list of sortal restrictions `SortalRestrictions` is not used otherwise within the ADT package but can be used during VIT construction.

Optional subgoals `Goals` are called when accessing a lexicon.

`vitValues.pl` contains predicates defining value ranges for designators as described in [Kasper et al. (2000)] or in the original databases.

Note, that **every lexicon access is deterministic**. I.e. once a matching entry was found no choice points remain and no other entries will be tried.

D.3 Format of a Linking Table

The linking tables are indexed in two ways: by semantic predicate names and syntactic lemmata. The format is

```
Language:vitSemLinking(PredName,Lemma,Particles,Stem,Linking) :-
   Subgoals.

Language:vitSynLinking(Lemma,Particles,Stem,PredName,Linking) :-
   Subgoals.
```

`Language` defines a language specific module.

`PredName` is the name of the predicate given by the semantic databases.

`Lemma` is also given by the semantic databases.[12]

`Particles` is a list of separable particles or prefixes belonging to a word stem `Stem`.

`Linking` is a list with elements of the form `arg(SemRole, Case, GramFunc, ArgType)` and with arguments as follows:

1. a semantic role `SemRole` or maybe a list of roles if the clause corresponds to an ambiguous verb;
2. a case `Case` or maybe a list of case values which corresponds to a list of grammatical functions, repectively. In case of a subcategorized preposition (`SemRole=arg4`), the value is always `noc`, for "no case";
3. a grammatical function `GramFunc` or maybe a list of function values which corresponds to a list of case values, repectively. In case of a subcategorized preposition, the preposition value or a list of possible ones. If no specific value can be determined, the default is `nof` for "no function";
4. an argument type `ArgType`. The default is `np` if in the original database does not specify a value.

For further value specifications see Section 1.4.4 or the original Prolog file.

[12] The +-signs marking separable prefixes/particles in ths original databases are removed in the on-line version.

D.3.1 Access to Linking Tables

The ADT package provides the following access predicates to the linking tables:

`vitGramRole(+Language,+Predicate,+GRole,-SArg)`

Given a language, the name of a semantic predicate and a grammatical role, `vitGramRole/4` returns the corresponding semantic argument role.

Example:

```
| ?- vitGramRole(ge,absagen,subj,Arg).

Arg=arg1

| ?- vitGramRole(ge,absagen,obj2,Arg).

Arg=arg2
```

D.4 Adding Further Lexicon Entries

The ADT packages comes with hook predicates which makes it very easy to add new entries to the on-line version. The supported languages are English, German, and Japanese whereas the lexicons can be found in modules `en`, `de`, and `jp`, respectively. All additional lexicon entries should have the format described in Section D.2.

First, each of these modules checks for lexical entries in a module named *language*`_first`, where *language* is one of `en`, `de`, and `jp` depending on the selected language. Then the original lexicon of the ADT package is looked up. Finally, each of these language modules checks for lexical entries in a module named *language*`_last`. Hence, on the one hand, one can add entries at the beginning of a lexicon using module *language*`_first` to overide existing entries. On the other hand, adding at the end of the existing lexicons is used to enlarge the lexicon. The named hook predicates are all assumed to be declared `multifile` and `dynamic`.

Beside these hooks, all user defined lexicon modules are considered when a user switches manually to a language using `vitSetMyLanguage/1` (see Section 5.3).

Note that both, the lexicon hooks and the user defined language modules, only extend the possible contents of the `Conditions` slot. The remaining information in a VIT is assumed to be known.

Index