Gefördert vom

bmb+f

Bundesministerium für
Bildung, Wissenschaft,
Forschung und Technologie

Verbmobil

Verbundvorhaben

The ADT Package for the Verbmobil Interface Term

Michael Dorna

Universität Stuttgart

Vm

Report 104
July 1996

July 1996

Michael Dorna

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

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

Gehört zum Antragsabschnitt: 7 Formalismus

Die vorliegende Arbeit wurde im Rahmen des Verbundvorhabens Verbmobil vom Bundesministerium für Bildung, Wissenschaft, Forschung und Technologie (BMBF) unter dem Förderkennzeichen 01 IV 101 U gefördert. Die Verantwortung für den Inhalt dieser Arbeit liegt bei dem Autor.

Abstract

This documentation describes an interface ADT called the "Verb*mobil* *Interface Term*" (VIT) used in the "Verb*mobil* Forschungsprototyp" (FP) in several software components. We present the contents of the VIT and the ADT package for Prolog components of the FP. Among others the ADT package can be used for creating, for manipulating, for printing and for checking the contents of a VIT.

The edition of this documentation corresponds to version 1.6.2 of the ADT package. An on-line HTML documentation is available via WWW at `http://www.ims.uni-stuttgart.de/projekte/verbmobil/vitADT`.

Contents

1 Introduction

1.1 The Verbmobil Background

Verbmobil is one of the largest projects in the area of machine translation. The main goal is a mobile translator for face-to-face dialogs, i.e. translation of spontaneous spoken language.

For the current implementation called "Verbmobil Forschungsprototyp" (FP) there are about 25 software components[1] under development. All these components run their own processes using channels for interprocess communication. Hence, Verbmobil can be seen as a large software development project and also a software engineering challenge.

We can distiguish the following parts of the FP. The *front end* includes components like Speech Recognition and Prosodic Labeling. The *"language" part* includes components like Syntactic-Semantic Analysis, Semantic Evaluation, Transfer and Generation. The *back end* includes components like Speech Synthesis and Speech Output.

The ADT is used in a uniform way between almost every component in the language part. Semantics play the crucial role in the language understanding and translation process of the FP. Therefore, most of the information within the ADT is concerned with semantics.

The main idea behind this approach is a single data structure for different "language" components. This lean data structure is portable to different programming languages used in the FP.

Christian Lieske, Joachim Quantz and myself started to define an ADT in September 1995 with the name *Minimally Recursive Structure*. In the meanwhile, a lot more Verbmobil partners of Generation, Semantic Construction, Semantic Evaluation, System Integration, Tempus and Aspect, and the Transfer were involved in the designing and development of the now renamed ADT: the Verbmobil *Interface Term*.

1.2 The Verbmobil Interface Term

As already mentioned, the VIT is used as a uniform data structure at the interfaces between several software components of the Verbmobil Forschungsprototyp. These

[1] We distinguish between software *components* and *modules*: a component is realized as one process which might include different software modules or control different modules which run their own processes.

interfaces are between Semantic Construction and Semantic Evaluation, Semantic Construction and Transfer as well as Transfer and Generation. The VIT is an encoding of different linguistically motivated information produced and used in the named components.

The contents of a VIT correspond to a segment (aka utterance) in a dialog 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 and prosody is also part of a VIT. This information is linked to semantics and can be used for the computing semantic tense, for disambiguation of underspecified analyses, for guiding semantic evaluation such as anaphora resolution and for many more.

There are different syntactic and semantic analysis modules realized in the FP which are based on both different linguistic theories and different formalisms for their implementation. At the interface to other components the syntax-semantic components map their output into the common ADT. Because all parties agree upon the interface, the differences are not transparent from outside.[2]

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. 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 protocol. Language-specific on-line dictionaries are used to insure the compatibility between components. A content checker is used to test structural properties. It has been shown that this form of protocol 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 protocol 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 and productive to improve a system than common, purely quantitative, measures based on failure or success rates.

[2] The project partners which are responsible for semantics even agree on the semantic analyses at the interface. This allows for a very flexible switching between the linguistic analysis components.

- The single protocol serves as a common "language" 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 Information Levels of a VIT

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

Slot Name	Description
Utterance ID	a unique tag for a segment or utterance of a turn the rest of the VIT belongs to;
Semantics	a list of labeled conditions describing the possibly underspecified semantic content of an utterance;
Main Label	the label of the main semantic condition, i.e. the entry point for traversing the semantic representation;
Sorts	a list of sortal information for marker variables introduced in labeled conditions;
Discourse	a list of additional semantic information, e.g. discourse roles for individuals introduced in labeled conditions;
Syntax	a list of morpho-syntactic information, e.g. case and gender of individuals;
Tense and Aspect	a list of morpho-syntactic and semantic tense combined with aspect information, e.g. used for computing surface tense;
Scope	a list of scope and grouping constraints, e.g. used for underspecified quantifier and operator scope representation;
Prosody	a list of prosodic information like accents and mood;
Groupings	a list of grouping constraints (belonging to Semantics).

A minimally recursive representation was chosen for efficient information access. The list arguments 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 arrays with fields containing constants.

1.4 Contents of VIT Slots

In this section we describe the information which can be found in the slots of a VIT. First, we list the terms,[3] and then we explain the possible argument bindings.

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

```
segment_description(ID, YesNo, LAtom)
```

`ID` is a unique Prolog atom for each VIT. `YesNo` is used to mark the last segment of a turn. `LAtom` is a Prolog atom of the utterance the rest of VIT belongs to. `LAtom` encodes a string using LaTeX conventions for German.

The contents of the Semantics slot are language specific. In general, it contains terms of the form

`Functor(Label, Arg2, ..., Arg`x`).`

called *labeled conditions*. The semantic entities are e.g. predicates, roles, operators, and quantifiers. The first argument 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 and operator scope, etc. is no longer syntactically represented in a recursive representation, 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 labeled conditions are all collected in databases, e.g. see [Heinecke et al. (1996)] for the German database. For further description of the semantics together with examples of VITs see [Bos et al. (1996)].

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

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

Info	Description	Slot Name
aktionsart(I, Art)	Aktionsart information	Tense and Aspect
cas(Inst, Case)	syntactic case	Syntax
ccom_plug(Hole,Label)	SynSem scope resolution	Scope
demontype(Inst, DType)	type of a demonstrative	Discourse
dialog_act(DialogAct)	SemEval dialog act information	Discourse
dialog_phase(DialogPhase)	dialog phase	Discourse
dir(Label, YesNo)	(non)directional preposition	Discourse
e_rel_r(Inst, Relation)	Reichenbachian tense relation	Tense and Aspect
eq(Label,Hole)	Label is equal to Hole	Scope
eval_plug(Hole,Label)	SemEval scope resolution	Scope
honor_inst(Inst)	politeness marker (for Japanese)	Discourse
honor_rel(Label)	politeness marker (for Japanese)	Discourse
gend(Inst, Gender)	morpho-syntactic gender	Syntax
leq(Label,Hole)	scope/subordination constraint	Scope
num(Inst, Number)	morpho-syntactic number	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_mood(Label, PMood)	prosodic mood	Prosody
r_rel_s(Inst, Relation)	Reichenbachian tense relation	Tense and Aspect
sem_group(L, ListOfLs)	group of conditions	Scope
syn_voice(Inst, Voice)	voice information (for Japanese)	Syntax
s_concept(Inst, Sort)	SemEval concept information	Sorts
s_sort(Inst, Sort)	sortal restriction	Sorts
ta_aspect(Inst, Aspect)	aspectual information	Tense and Aspect
ta_mood(Inst, TMood)	mood	Tense and Aspect
ta_tense(Inst, Tense)	surface tense	Tense and Aspect
tmod(Inst, TMod)	temporal modification	Tense and Aspect
unbound(Label)	unbound argument	Scope

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

Argument	Description or list of values
`Art`	`acc`, `ach`, `act`, `stat`
`Arg`x	`Hole`, `Inst`, `Label`, `Atom`, or a Prolog list of the named `Args`
`Aspect`	`progr`, `nonprogr`
`Atom`	Prolog atomic
`Case`	`nom`, `gen`, `dat`, `acc`
`Check`	`ground`, `shape`
`Class`	`ambig`, `aux`, `disc`, `grad`, `mod`, `mood`, `noun`, `prep`, `pron`, `quant`, `verb`
`DialogAct`	Prolog atom
`DialogPhase`	Prolog atom
`DType`	`near`, `far`, `ident`, `spec`
`Functor`	Prolog functor
`Gender`	`fem`, `masc`, `neut`
`Hole` or `H`	Prolog atom starting with `h` or `ht`followed by an unsigned integer
`Info`	any valid term encoding information in a VIT
`Inst` or `I`	Prolog atom starting with `i` or `it` followed by an unsigned integer
`ID`	Prolog atomic
`Label` or `L`	Prolog atom starting with `l` or `lt` followed by an unsigned integer
`Language`	`de`, `en`, `jp`
`LAtom`	Prolog atom encoding a LaTeX string
`ListOfLs`	Prolog list of `Label` elements
`TMood`	`ind`, `conj`, `imp`
`Number`	`sg`, `pl`
`Person`	`1`, `2`, `3`
`PRef`	`sp`, `he`, `sp_he`, `third`, `top`
`PType`	`refl`, `std`, `refl_std`, `recip`, `imp`, `event`, `event_std`, `demon`, `demon_event`, `zero` (and `intersent` for Japanese)
`PMood`	`decl`, `prog`, `quest`
`Relation`	`equal`, `overlap`, `follow`, `precede`
`SMood`	`decl`, `imp`, `ynq`, `ynq_imp`, `whq` and `ynq_decl` (for Japanese)
`SlotName`	`'Groupings'`, `'Scope'`, `'Tense and Aspect'`, `'Main Label'`, `'Prosody'`, `'Discourse'`, `'Semantics'`, `'Sorts'`, `'Utterance ID'`, `'Syntax'`
`Sort`	sort expression: `Sort;Sort` or `Sort&Sort` or `~Sort` or `Atom`
`TMod`	`st_dist`, `st_equ`, `st_prec`, `st_perf`, `st_quant`
`Tense`	`infin`, `plusq`, `perf`, `praet`, `pres`, `prespart`, `futI`, `futII` (for German and Japanese) `infin`, `pres`, `pastperf`, `past`, `future`, `futurepast`, `presperf` (for English)
`Voice`	`act`, `pass`
`VIT`	Verb*mobil* Interface Term (not necessarily ground)
`YesNo`	`yes`, `no`

1.5 The Prolog Implementation

In Prolog, the VIT is implemented as a term of arity 10 named `vit`. The name and arity of this realization should be of no interest, if the access into the 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. The elements are terms. This data structure is very flexible in adding and removing information, i.e. the terms.

The input and output of each component dealing with the VIT has to make sure the following properties:[4]

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

[4] Of course, all partners are free to use any other representations *within* their components.

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 VIT contents. 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 is part of the ADT package distribution. Finally, appendix D sketches briefly the contents of the on-line dictionaries given by the files `vitSemLex.pl` and `vitValues.pl` of the distribution.

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 the ADT

2.1 Introduction

With `vitNew/1` a new VIT can be created, and with `vitAdd/2` it can be filled with content. In this case, the adding of information is realized by open ended lists which should be closed finally using `vitClose/1`.

Once the open ended lists are closed `vitAdd/3` can be used to add new information to a VIT. `vitAdd/3` takes a VIT, adds something and results a new VIT.

For rebuilding a given VIT which might be not compatible with the current format `vitRebuild/2` produces a new one.

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 5.4).

2.2 Predicates

vitNew(-VIT)

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

Example:

```
| ?- vitNew(VIT).

VIT = vit(_099,_100,_101,_102,_103,_104,_105,_106,_107,_108)
```

vitAdd(+Infos, ?VIT)

Adds information to the contents of the a VIT. `Infos` can be a single `Info` or a list of `Info` elements. An utterance id `UID` can be added using `id(UID)` for `Info`. In the same way, the Main Label slot can be filled with a label `Label` using `main_label(Label)`. The behaviour of `vitAdd/2` can be influenced using `vitLazyCheck/0` (see section 5.4).

Example:

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

VIT = vit(_88,_89,_90,[s_sort(i1,top)|_105],_92,_93,_94,_95,_96,_97)
```

vitAdd(+Info, +VIT, ?VIT)

Adds a single information `Info` to the contents of a VIT resulting a new or modified VIT. The behaviour of `vitAdd/3` can be influenced using `vitLazyCheck/0` (see section 5.4).

Example:

```
| ?- vitNew(VIT),
     vitAdd([id(xyz),main_label(l1)],VIT),
     vitClose(VIT),
     vitAdd(s_sort(i1,top),VIT,VIT1).

VIT = vit(xyz,[],l1,[],[],[],[],[],[],[]),
VIT1 = vit(xyz,[],l1,[s_sort(i1,top)],[],[],[],[],[],[])
```

vitClose(?VIT)

Closes the arguments of a VIT which may contain open ended lists.

Example:

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

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

vitRebuild(+VIT,-VIT)

Produces a new VIT using the information found in a given VIT. The new one may be different with respect to the location of information. Both VITs should have arity 10 and should have the Utterance ID and Main Label slot at the same argument position.

3 Access the ADT

3.1 Introduction

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 terms* within the a VIT.

The slot access predicates have the form `vit*(?VIT,?SlotContent)` and can be used to manipulate a slot directly. In general, the term 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. Otherwise, they fail.

3.2 Accessing the Slots

`vitDiscourse(?VIT, ?Discourse)`

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

`vitGroupings(?VIT, ?Groupings)`

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

`vitID(?VIT, ?UtteranceID)`

Unifies `UtteranceID` with the Utterance ID slot of a VIT.

`vitMainLabel(?VIT, ?MainLabel)`

Unifies `MainLabel` with the Main Label slot of a VIT.

`vitProsody(?VIT, ?Prosody)`

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

`vitScope(?VIT, ?Scope)`

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

`vitSemantics(?VIT, ?Semantics)`

Unifies `Semantics` with the Semantics 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.

3.3 Accessing the Terms

Note: In general, the following predicates fail if the required information is not part of the VIT used in a call.

`vitAktionsart(+Inst, +VIT, ?Art)`

Reports the Aktionsart value for a given instance.

`vitAmbig(+VIT, -Ambiguities)`

Reports all labeled conditions in the Semantics slot which belong to one of the lexical ambiguity classes (see [Bos et al. (1996)]). `Ambiguities` is a list of such conditions or `[]` if there are none.

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

Reports the aspect value for a given instance.

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

Reports the case value for a given instance.

`vitConcept(+Inst, +VIT, ?Concept)`

Reports the concept value for a given instance.

`vitDemonType(+Inst, +VIT, ?DType)`

Reports the type of a demonstrative with instance `Inst`.

`vitDialogAct(+VIT, ?DialogAct)`

Reports the the dialog act of a VIT.

`vitDialogPhase(+VIT, ?DialogPhase)`

Reports the the dialog phase of a VIT.

`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`).

`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).

`vitErelR(+Inst, +VIT, ?Relation)`

Reports a tense relation for a given instance.

`vitGroup(?Label,+VIT,-ListOfLs)`

Looks for a single grouping (backtrackable).

`vitLabelInGroup(+Label,+VIT,-GroupLabel,-ListOfLs)`

Checks for a given label `Label` if it is a member of a `ListOfLabels` and requires `vitGroup(GroupLabel,VIT,ListOfLabels)`.

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

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

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

Reports the person value for a given instance.

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

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

`vitProsAccent(+Label, +VIT)`

Checks if there is a prosodic accent 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`.

`vitRrelS(+Inst, +VIT, ?Relation)`

Reports a tense relation for a given instance.

`vitSegmentDesc(?VIT, ?ID, ?YesNo, ?LAtom)`

Unifies a `segment_description/3` term (see section 1.4).

`vitSemEvalPlug(+Hole,+VIT,-Label)`

Reports the plugging for a hole suggested by Semantic Evaluation. Checks for `eval_plug/2`. If it is not available, it tests `vitSynSemPlug(Hole,VIT,Label)`.

`vitSentMood(+VIT, ?SMood)`

Reports the sentence mood for a given `VIT`. `SMood` is the functor of the condition with label `MainLabel` and restricted to the sentence mood values given in section 1.4.

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

Reports the sort value for a given instance.

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

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

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

Reports the (morpho-syntactic) gender value for a given instance.

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

Reports the (morpho-syntactic) number for a given instance.

`vitSynSemPlug(+Hole,+VIT,-Label)`

Reports the plugging for a hole suggested by Semantic Construction, e.g. based on a syntactic analysis.

`vitTMod(+Inst, +VIT, ?TMod)`

Reports the temporal modifier value for a given instance.

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

Reports the (morpho-syntactic) tense value for a given instance.

`vitVoice(+Inst, +VIT, ?Voice)`

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

4 Copy and Delete Information

4.1 Introduction

`vitCopy/2` makes a copy of a given VIT which shares no variables with the original. `vitDelete/3` deletes information in a given VIT. `vitCopyAllBut/3` makes a copy of a VIT without copying a specified slot.

4.2 Predicates

`vitCopy(+VIT, -VIT)`

Makes a copy of a VIT.

Example:

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

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

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

Makes a copy of all slots of a VIT without copying argument `Integer`. This predicate should always be used in combination with `vitSlotName/2` (see section 7).

Example:

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

No = 4,
VIT = vit(A,[],B,[s_sort(i1,top)],[],[],[],[],[],[]),
NewVIT = vit(A,[],B,_8300,[],[],[],[],[],[])
```

`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,top)],[],[],[],[],[],[]),
               VIT).

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

5 Check the VIT

5.1 Introduction

In this section all predicates are listed which can be used to check the VIT or parts of it.

`vitCheckFormat/1` checks a VIT syntactically whereas `vitCheckContent/1` does the same for dependencies between the pieces of information in a VIT. `vitCheck/1` combines both predicates.

An error handler reports the results of the ADT checkers. 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`. The action after some error(s) have been detected and reported can be chosen using `vitIgnoreError/0` or `vitFailOnError/0` (default).

`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. The system will report every detection of unknown information during processing but will not fail. To switch off this mode use `vitRegularCheck/0` (default).

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`, and `vitHole/1`.

The language of the checker can be switched using `vitSetLanguage/1`. The current language can be seen be using `vitLanguage/1`.

5.2 VIT Checkers

`vitCheck(+VIT)`

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

`vitCheckFormat(+VIT)`

Checks the syntax of the information in a given VIT. I.e. for each single `Info` the compatibility with the semantic lexicon (e.g. argument frames for verbs) of the current ADT language is checked and also the argument value ranges of the `Info` terms.

`vitCheckContent(+VIT)`

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

- unique concept/sort/tense/mood assignment to instances;
- existence of type information for pronouns and demonstratives;
- existence of (non-)directional information for prepositions;
- existence and uniqueness of `leq/2`, `ccom_plug/2` and `eval_plug/2` entries for a hole;
- unique plugging for a single label;
- detection of cyclic groupings;
- missing groupings for group labels;
- existence of morpho-syntactic tense and mood for verbs.

Further checking for cycles and connections between information will be added in future.

5.3 Error Handling

`vitReportErrors`

Reports all detected errors after checking (default).

`vitDontReportErrors`

Reports no errors even if some were detected during checking.

`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 accessable and writeable file or a Prolog output stream. Default is `user_error`.

`vitResetErrorOutput`

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

5.4 Information Checking

`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).

`vitSetLanguage(+Language)`

Switches the ADT to a given language.

`vitLanguage(?Language)`

Reports current language setting.

`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,top),ground,'Semantics').

no
| ?- vitValidInfo(s_sort(i1,top),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. This predicate is **not** exported by `vitADT`!

Example:

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

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

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.

6 Printing

6.1 Introduction

vitPrint/{1,2} pretty-print a VIT such that it can be used as a Prolog term again.

6.2 Predicates

vitPrint(+VIT)

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

Example:

```
vit( segment_description(tb1t1br1u1,yes,
                         'dann machen wir doch noch einen termin aus'),
     [ausmachen(l6,i1),                % Semantics
      doch(l9,h3),
      termin(l8,i2),
      decl(l7,h4),
      arg1(l6,i1,i3),
      arg3(l6,i1,i2),
      noch_fadv_padv(l5,l15,h2),
      ein_card_qua(l4,i2,l14,l1,h1,1),
      dann_laprep_padv(l3,i1,h5,l16,i4,h4,l2,spec),
      pron(l12,i3)],
     l7,                               % Main Label
     [s_sort(i1,ment_communicat_poly), % Sorts
      s_sort(i2,&(space_time,time_sit_poly)),
      s_sort(i3,&(human,person))],
     [dir(l3,no),                      % Discourse
      prontype(i3,sp_he,std)],
     [num(i3,pl),                      % Syntax
      pers(i3,1),
      gend(i2,masc),
      num(i2,sg),
      pers(i2,3),
      cas(i2,acc),
      cas(i3,nom)],
     [ta_mood(i1,ind),                 % Tense and Aspect
      ta_tense(i1,pres)],
     [unbound(l16),                    % Scope
      unbound(l15),
      unbound(l14),
      leq(l2,h4),
      leq(l2,h3),
      leq(l2,h2),
      leq(l2,h1),
```

```
          leq(l9,h4),
          leq(l5,h4),
          leq(l4,h4),
          ccom_plug(h5,l9),
          ccom_plug(h4,l3),
          ccom_plug(h3,l5),
          ccom_plug(h2,l4),
          ccom_plug(h1,l2)],
         [pros_mood(l7,decl)],               % Prosody
         [sem_group(l2,[l6]),                % Groupings
          sem_group(l1,[l8])]
       )
```

vitPrint(+Stream,+VIT)

Pretty-prints a VIT directing the output to a given stream.

7 Miscellaneous

7.1 Introduction

This part describes further predicates not covered by previous sections.

`vitLabelCond/3` constructs or decomposes a labeled condition.

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

`vitPredicate/3` and `vitExistsPred/4` can be used to check for predicates. `vitExistsCond/3` can be used to check for any condition restricted by a class and is similar to `vitCondition/4`.

`vitInstInfo/3` can be used to collect all information about an instance.

`vitDisambig/4` can be used to expand underspecified lexical items.

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

7.2 Predicates

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

Decomposes a given labeled 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)
```

vitSlotName(?Name,?Integer)

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

Example:

```
| ?- vitSlotName('Main Label',I).

I = 3
```

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 labeled 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. Notice: No arg$X$/3, X_arg/3, etc. and no elements of class aux (like perf/3) will be matched!

vitInstInfo(+Inst,+VIT,-Infos)

Collects all information about an instance into a list of Attribute:Value pairs for a given VIT.

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

Checks whether two labels are connected by a transitive closure $\stackrel{*}{\geq}$ over labeled conditions and scoping constraints (backtrackable). Given a label `Label1/Label2` this predicates enumerates all labels `Label2/Label1` for which `Label1` $\stackrel{*}{\geq}$ `Label2` holds. This closure is restricted by scopal islands given by modal verbs as well as sentential complements.

`vitDisambig(+Relation,+SemClass/PredName,+VITin,-VITout)`

Given an underspecified semantic representation `Relation`, a specific semantic class `SemClass` or predicate name `PredName` (for verbs) and a VIT, this predicate produces a VIT in which the underspecification is replaced by an instance of the specified class. `Relation` should be part of the given VIT, of course.

`vitInit`

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

`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:
% vitRegularCheck
% vitIgnoreError
% vitReportErrors
% vitErrorOutput(user_error)
% vitLanguage(de)
```

References

[Amtrup (1995)] Jan W. Amtrup. *ICE - INTARC Communication Environment. Users Guide and Reference Manual. Version 1.3.* Verbmobil Technisches Dokument 14, Universität Hamburg, Oktober 1995.

[Bos et al. (1996)] Johan Bos, Markus Egg and Michael Schiehlen. *Definition of the Abstract Semantic Classes for the Verbmobil Forschungsprototyp 1.0.* Draft of July 11, 1996. Available via WWW `http://coli.uni-sb.de/~vm/vm-internal/vitdocu.ps.gz` (access restricted to Verbmobil partners).

[CoLi (1996)] Karsten Worm (ed.). *Japanese Semantic Database.* Version of July 23, 1996. Available via WWW `http://coli.uni-sb.de/~vm/vm-internal/vm-japan.html` (access restricted to Verbmobil partners).

[CSLI/DFKI/IAI/IBM/IMS/SfS (1996)] Wolfgang Finkler (ed.) *English Semantic Database.* Version 1.9 of July 23, 1996. Available via WWW `http://www.dfki.uni-sb.de/~finkler/E-SEMDB.rdb` (access restricted to Verbmobil partners).

[Heinecke et al. (1996)] Johannes Heinecke and Karsten Worm. *Semantische Datenbank.* Version 1.0, patch of July 24, 1996. Available via WWW `http://www.compling.hu-berlin.de/~vm/semdb/semdb-aktuell.gz` (access restricted to Verbmobil partners).

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

A Getting and Installing the Software

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
ftp> bin
ftp> get vitADT1.6.2.tar.gz
```

Afterwards you have to unzip and untar the file at your local site, e.g. using

```
% gtar -zxf vitADT1.6.2.tar.gz
```

or

```
% unzip vitADT1.6.2.tar.gz; tar -xf vitADT1.6.2.tar
```

The distribution contains the following files:

`README`	release notes (recent changes) and installation guide;
`Makefile`	the makefile;
`code/vitSemlex.pl`	the built-in semantic lexicon (see appendix D);
`code/vitValues.pl`	value ranges of terms (see appendix D);
`bin/vitADT.qof`	ADT package in module `vitADT` (with the semantic lexicon);
`bin/atom2term.qof`	conversion of terms to atoms and vice versa (see appendix C);
`doc/report-104-96.ps`	this document.

To install the software call

```
% make install
```

which copies `vitADT.qof` to `$VM_HOME/etc/Vit_Adt`. For a regular Verb*mobil* Forschungsprototyp installation you should set the `VM_HOME` variable in `Makefile` or the environment.

To run the software you neeed Quintus Prolog [Quintus Manual] Version 3.2 or higher.

B Usage

B.1 Using the Package

The name of the ADT package is `vitADT.{pl,qof}`. 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, ListOfImportedPredicates)`.

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 ICE [Amtrup (1995)] 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 `'$VAR'`(*Integer*) terms. These `'$VAR'/1` might be invisible during debugging 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 Built-in Semantic Lexicon Database

This ADT package contains the files `vitSemLex.pl` and `vitValues.pl` which define an on-line database. Currently, there exists an on-line lexicon for German (based on [Heinecke et al. (1996)]), for English (based on [CSLI/DFKI/IAI/IBM/IMS/SfS (1996)]) and for Japanese (based on [CoLi (1996)]). All three are based on the semantic class descriptions of [Bos et al. (1996)] except for the changes and reductions mentioned below.

We give a brief description of the data which is part of the current ADT package. The data 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 it does not contain the whole information of the original, e.g. for decomposition classes which are part of the German database. On the other hand, we have changed some semantic classes and their names as follows:

old suffix	new name	VIT representation
iv	arg1	`R(L,I),arg1(L,I,I1)`
iv	arg3	`R(L,I),arg3(L,I,I1)`
tv	arg12	`R(L,I),arg1(L,I,I1),arg2(L,I,I2)`
tv	arg13	`R(L,I),arg1(L,I,I1),arg3(L,I,I2)`
tv	arg23	`R(L,I),arg2(L,I,I1),arg3(L,I,I2)`
dv	arg123	`R(L,I),arg1(L,I,I1),arg2(L,I,I2),arg3(L,I,I3)`
ipcv	pcv_arg3	`R(L,I),arg3(L,I,H)`
pcv	pcv_arg13	`R(L,I),arg1(L,I,I1),arg3(L,I,H)`
pcv	pcv_arg23	`R(L,I),arg2(L,I,I1),arg3(L,I,H)`
dpcv	pcv_arg123	`R(L,I),arg1(L,I,I1),arg2(L,I,I2),arg3(L,I,H)`
coord	coord_s	`R(L,LH,LH1)`
coord	coord_c	`R(L,I,LH,I1,L2,I2)`
coord	coord_2h	`R(L,I,H,I1,I2)`
dra	dra_conj	`R(L,I,H,H1)`
dra	dra_mod	`R(L,I,H)`
sprep	prep	`R(L,I,I1)`

Some information was added manually which was not part of the databases, e.g. this information was (still) forgotten. Contextual constrains such as grouping, subordination, etc. are **not** part of this database. In general, these are the result of a semantic construction process, i.e. that's part of actions combined to phrase structure rules.

The database has the following form:

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

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

The first argument is always a labeled condition `LabeledCondition`.

In general, the semantic class `SemanticClass` is given by [Bos et al. (1996)]. Exceptions are the named classes mentioned above.

The syntax checking code `ConditonCheckingCode` and `ContextCheckingCode` is used, e.g., by `vitCheckFormat/1`. The callable predicates are either Prolog built-in or 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`.

The list of sortal restrictions `SortalRestrictions` is extracted from the German database. This information is not used in the ADT package.

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

The `vitValues.pl` file contains predicates defining value ranges for designators as described in [Bos et al. (1996)] or in the original databases.

Index