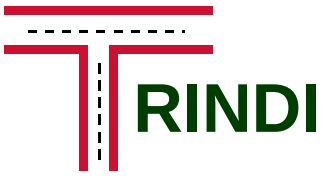
TRINDIKIT 1.0 Manual

Staffan Larsson, Peter Bohlin, Johan Bos, David Traum

Distribution: PUBLIC



Task Oriented Instructional Dialogue
LE4-8314

Deliverable D2.2 — Manual

November 1999

Task Oriented Instructional Dialogue

Gothenburg University
Department of Linguistics

University of Edinburgh
Centre for Cognitive Science and Language Technology Group, Human Communication Research Centre

Universität des Saarlandes
Department of Computational Linguistics

SRI Cambridge

Xerox Research Centre Europe

For copies of reports, updates on project activities and other TRINDI-related information, contact:

The TRINDI Project Administrator
Department of Linguistics
Göteborg University
Box 200
S-405 30 Gothenburg, Sweden
trindi@ling.gu.se

Copies of reports and other material can also be accessed from the project's homepage, http://www.ling.gu.se/research/projects/trindi.

Contents

Chapter 1

Introduction

This is a manual for the TRINDIKIT, a toolkit for building and experimenting with *dialogue move engines* and *information states*, that has been developed in the TRINDI project. We use the term *information state* to mean, roughly, the information stored internally by an agent, in this case a dialogue system. A *dialogue move engine* updates the information state on the basis of observed dialogue moves and selects appropriate moves to be performed.

Apart from proposing a general system architecture, the TRINDIKIT also specifies formats for defining information states, update rules dialogue moves and associated algorithms. It further provides a set of tools for experimenting with different formalizations of implementations of information states, rules, and algorithms. To build a dialogue move engine, one needs to provide definitions of update rules, moves and algorithms, as well as the internal structure of the information state. One may also add inference engines, planners, plan recognizers, dialogue grammars, dialogue game automata etc.. More information on information states and related concepts is available in Traum *et al.* (1999).

The DME forms the core of a complete dialogue system. Simple interpretation, generation, input and output modules are also provided by the TRINDIKIT, to simulate a end-to-end dialogue system. Examples of theories and systems implemented using the TRINDIKIT can be found in Traum *et al.* (1999) and Bos *et al.* (1999).

Note that TRINDI deliverable D2.2 is the actual TRINDIKIT implementation[1], which is available from the TRINDI web page `www.ling.gu.se/research/projects/trindi/`. This manual is a supplementary documentation to the actual deliverable. It should be noted that the TRINDIKIT, including the manual, is under development. Up-to-date

[1] The current TRINDIKIT implementation is written in SICSTUS prolog.

versions of the TRINDIKIT and manual are available from the TRINDI web page.

1.1 The TRINDIKIT architecture

In this section, we give an overview of the TRINDIKIT architecture, mention some central concepts, and discuss optional vs. obligatory components as well as toolkit provided vs. user provided functionality.

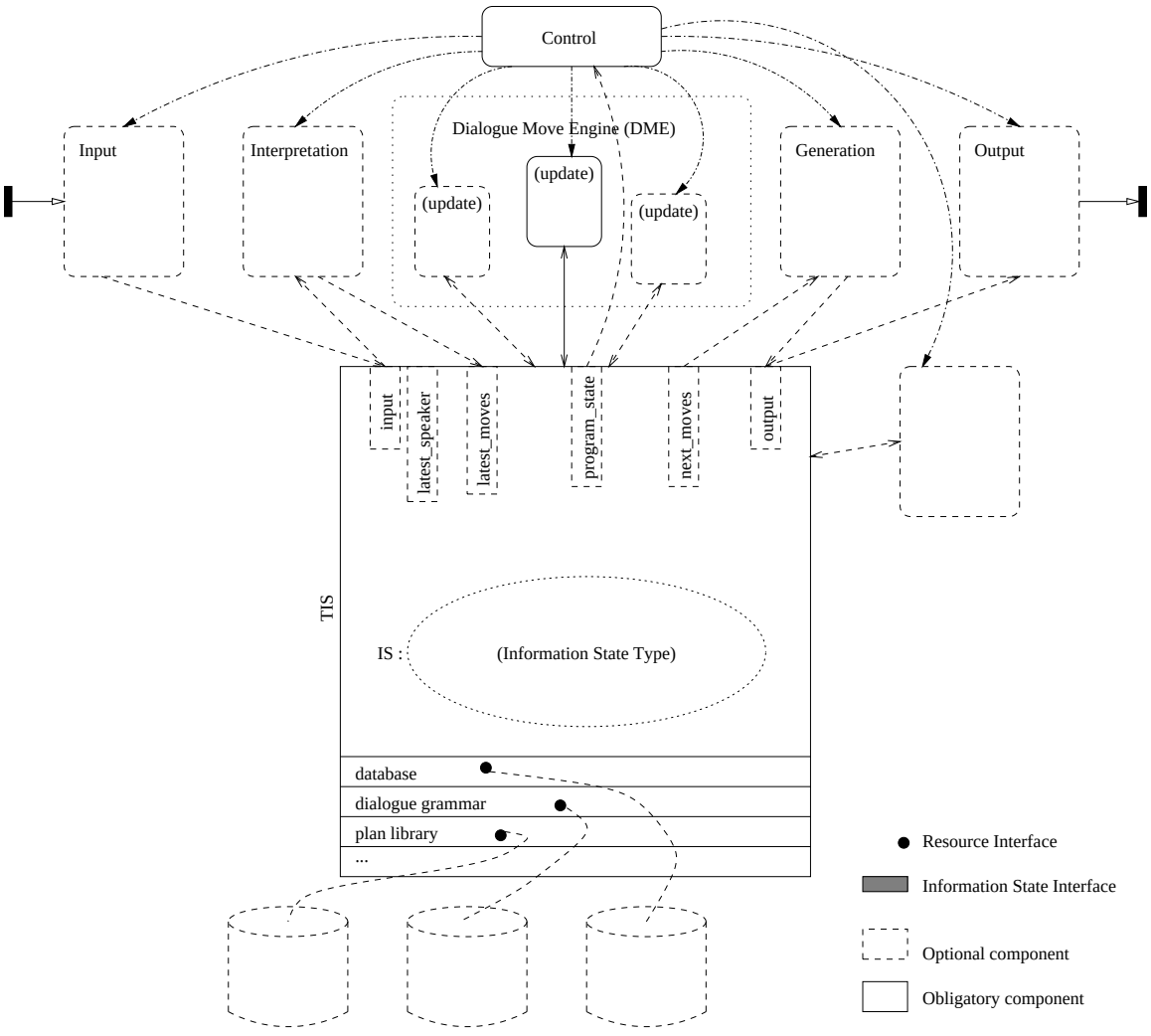


Figure 1.1: The TRINDIKIT architecture

The general architecture we are assuming is shown in Figure 1.1.

The component of the architecture are the

- the Total Information State (TIS), consisting of
  - the Information State (IS)
  - module interface variables
  - resource interfaces
- the Dialogue Move Engine, consisting of one or more DME modules
- non-DME modules
- a control module, wiring together the other modules, either in sequence or through some asynchronous mechanism.

Some of the components are obligatory, and others are optional or user-defined. To build a system, one must minimally supply

- An information state type.
- At least one DME module, consisting of TIS update rules and an algorithm.
- A control module, operating according to a control algorithm

Any useful system is also likely to need

- Additional modules, e.g. for getting input from the user, interpreting this input, generating system utterances, and providing output for the user.
- Interface variables for these modules, which are designated parts of the TIS where the modules are allowed to read and write according to their associated TIS access restrictions.
- Resources such as databases, plan libraries etc. The resources are accessible from the modules through the resource interfaces, which define applicable conditions and (optionally) operations on the resource.

One possible setup of non-DME modules, indicated by dashed lines in figure 1.1, is the following:

- Input: Receives input utterances from the user and stores it in the input interface variable.
- Interpretation: Takes utterances (stored in input) and gives interpretations in terms of dialogue moves (including semantic content). The interpretation is stored in the interface variable latest_moves.
- Generation: Generates output moves based on the contents of next_moves and passes these on to the output interface variable.
- Output: Produces system utterances based on the contents of the output variable

Corresponding to these modules, one could have the following interface variables:

- input: the input utterance(s)
- latest_moves: list of moves currently under consideration
- latest_speaker: the speaker of the latest move
- next_moves: list of moves for system's next turn
- output: the system's output utterance(s)
- program_state: current state of the DME (used for control)

A possible setup of the DME is to divide it into two modules, one for updating the TIS based on the latest move and one for selecting the next move:

- Update: Applies update rules to the DIS according to the update engine algorithm (also specified in SIS)
- Selection: Selects dialogue move(s) using the selection rules and the move selection algorithm. The resulting moves are stored in next_moves.

Note that the illustrated module setup is just an example. The TRINDIKIT provides methods for defining any number of both DME-modules and non-DME modules, with associated interface variables.

1.2 Toolkit provided functionality

Apart from the general architecture defined above, the TRINDIKIT provides

- definitions of datatypes, for use in TIS variable definitions
- a language and format for specifying TIS update rules
- methods for accessing the TIS
- an algorithm definition language for DME and control modules
- default modules for input, interpretation, generation and output
- methods for converting items from one type to another
- methods for visually inspecting the TIS
- debugging facilities

Chapter 2

TRINDIKIT Concepts

In this chapter, we review the central concepts in the TRINDIKIT architecture and relate them to each other.

2.1 Information State

The IS has the following basic characteristics:

- It is an object of a certain type
- The type of the IS determines the conditions and operations which can be applied to it
- It can be accessed (read from and written to) by modules

2.2 Dialogue Moves

Dialogue moves are moves (or acts, or actions) associated with utterances. A single utterance may be associated with several moves, simultaneous or ordered in time. In the TRINDI approach, dialogue moves are also related to information state updates.

2.3 Update rules

Update rules are rules for updating the information state (or more generally, the TIS). They consist of a rule name, a precondition list, and an effect list. The preconditions are conditions on the TIS, and the effects are operations on the TIS. If the preconditions of a rule are true for the TIS, then the effects of that rule can be applied to the TIS. Rule also have a class. Rules may be ordered in class hierarchies.

2.4 Update and control algorithms

Update algorithms are algorithms for updating the TIS. They include conditions on the TIS and calls to apply (classes of) update rules. Update algorithms are executed by DME modules, or other modules implemented using the TRINIDKIT facilities for specifying update rules and algorithms.

Control algorithms are similar to update algorithms, but instead of calling rule classes they call modules. The control algorithm is executed by the control module.

2.5 Modules

Modules have the following basic characteristics:

- they are processes
- they can read and/or write to the total information state
- they are called by the control module[1], and can thus be run asynchronously with other modules
- they can not be called from update rules

[1] Of course, this does not apply to the control module itself.

2.5.1 DME modules

DME modules are responsible for updating the information state based on observed moves, and for selecting moves. They consist of a set of update rules and an algorithm. All DME modules have access to the whole TIS.

A note on the difference between modules and resources: Resources are declarative knowledge sources, external to the information state, which are used in update rules and algorithms. Modules, on the other hand, are processes which interact with the information state and are called upon by the control module. Of course, there is a procedural element to all kinds of information search, which means among other things that one must be careful not to engage in extensive time-consuming searches. Conversely, modules can be defined declaratively and thus have a declarative element. There is no sharp distinction dictating the choice between resource or module; for example, it is possible to have the parser be a resource. However, it is important to consider the consequences of choosing to see something as a resource or module.

2.5.2 Non-DME modules

Non-DME modules have limited access to the TIS; each such module must explicitly declare which parts of the TIS it is allowed to read and write. Non-DME modules can be implemented in various ways, either using update rules and an algorithm or in other way, as the user sees fit.

2.5.3 Module interfaces

Usually, non-DME modules can only access a certain number of designated TIS variables, so-called *module interface variables*. The purpose of these variables is to enable non-DME modules to interact with each other and with the DME modules. It is possible to allow non-DME modules to access the IS, but this will significantly reduce the ability to use the module in systems using other kinds of IS types.

2.6 Resources and resource interfaces

Resources have the following basic characteristics:

- They are knowledge sources which can be dynamic or static
- They are called from update rules vi conditions and operations
- They are attached to the TIS via resource interfaces, consisting of definitions of conditions and operations on the resource
- They cannot read and write to the information state

The values of these variables are pointers to resources.

2.7 Total Information State

The Total Information State (TIS) consists of three components: the information state variable, the module interface variables, and the resource interface variables. These variables are also called *TIS variables*.

2.8 Flags

The flags are static while the system is running. They can be read by all modules but cannot be written to except by the user or in the system configuration file. There is a set of generic TRINDIKIT flags, but the user can add additional flags. All flags have a default value.

Chapter 3

Contents of the TRINDIKIT package

In this section we will give an overview of the structure of the TRINDIKIT implementation and any system implemented within the TRINDIKIT architecture. The figures presented here may look extremely complex and terrifying, but we hope that (1) they will make the relation between the implementation and the general TRINDIKIT architecture visible, and (2) they will provide a reference and overview useful for understanding the rest of this chapter.

3.1 Modular structure

The current TRINDIKIT is implemented in SICSTUS prolog, and uses the "module" facility which allows dividing an implementation into modules which export and import predicates to and from other modules. Any dialogue system built using the TRINDIKIT will have a modular structure similar to that in Figure 3.1.

Note that the prolog notion of modules has nothing to do with the TRINDIKIT notion of modules.

3.2 File structure

Any module may consist of one or several files. In Figure 3.2 the relation between the files of a TRINDIKIT dialogue system is shown.

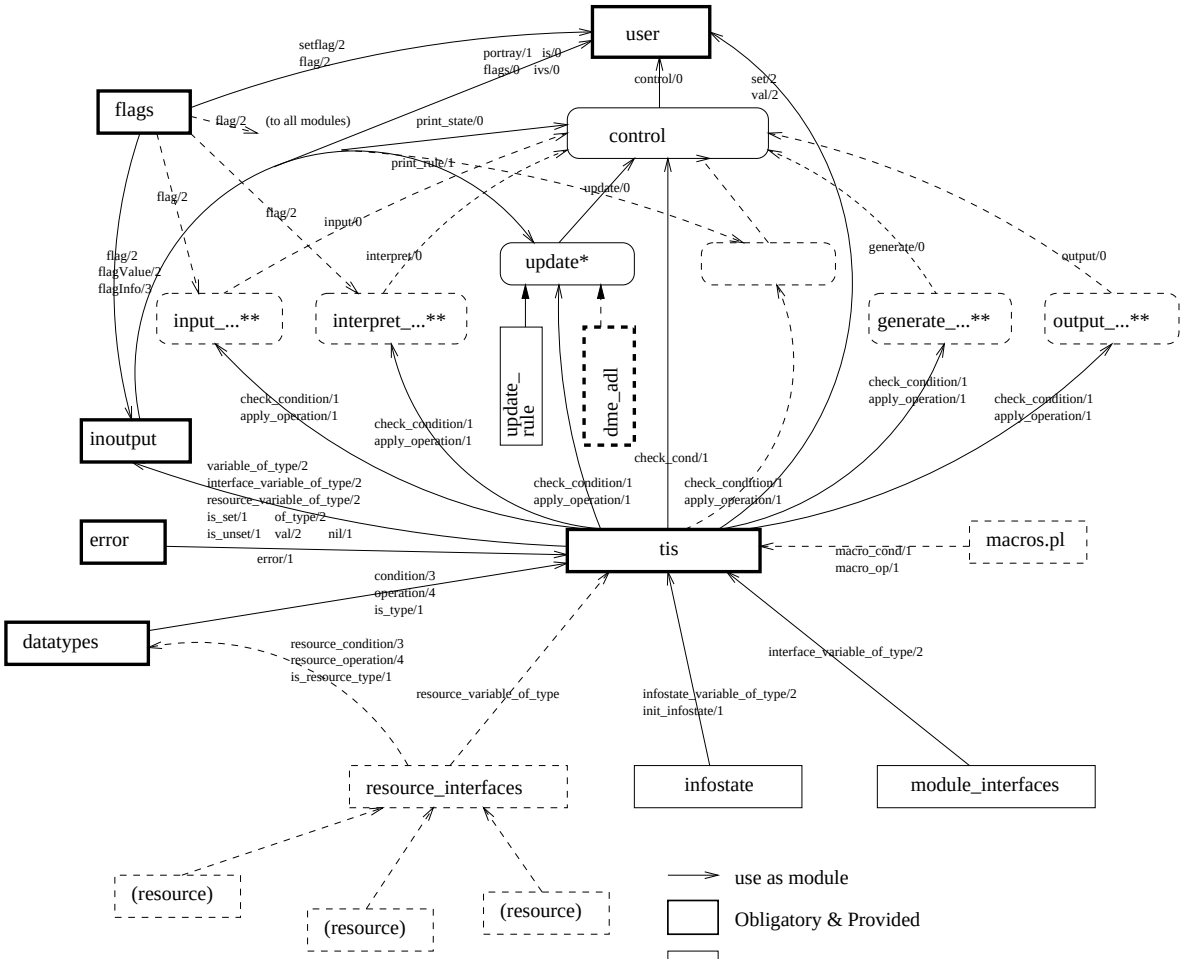


Figure 3.1: The TRINIDKIT module structure

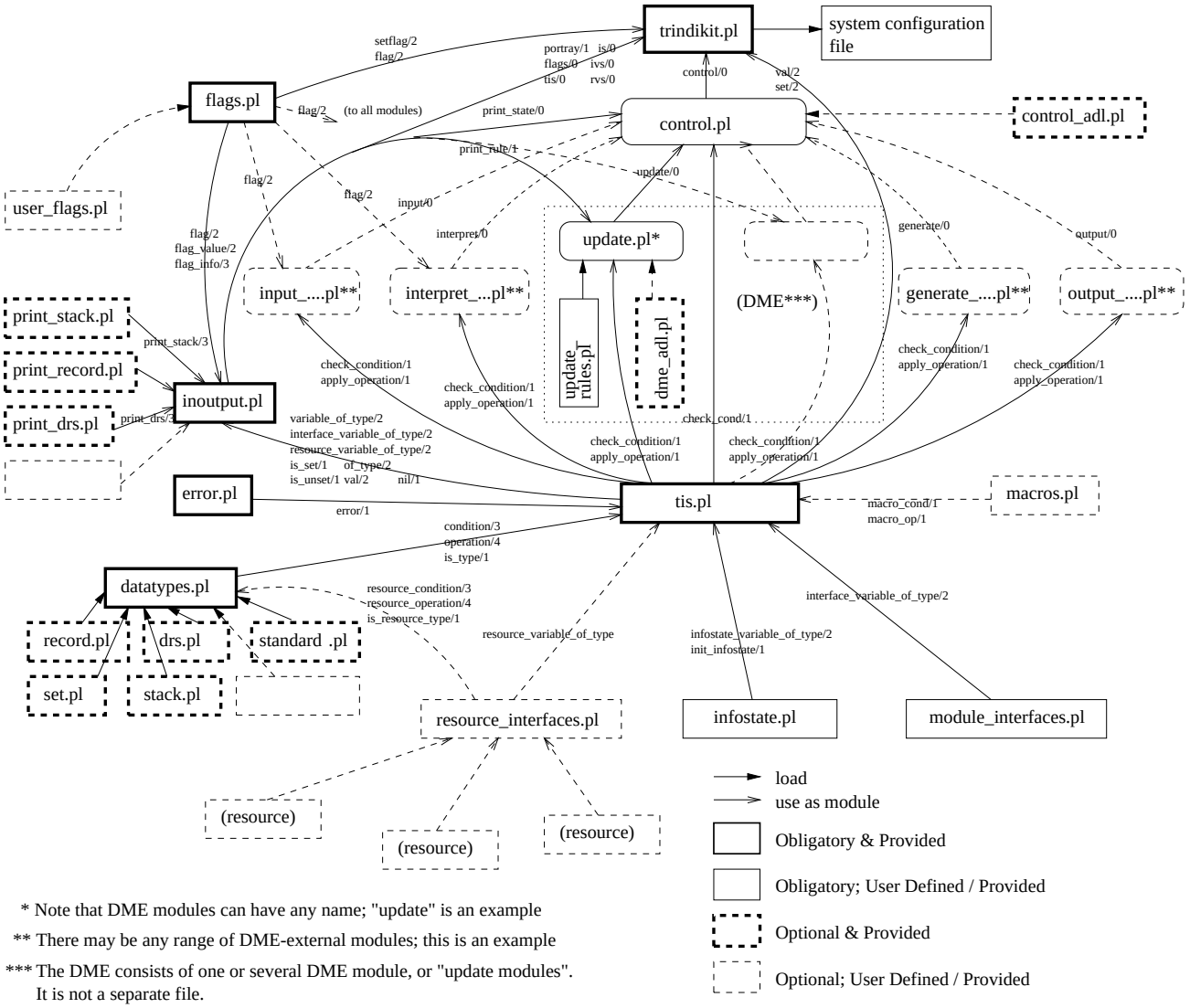


Figure 3.2: The TRINDIKIT file structure

3.3 TRINDIKIT files

The TRINDIKIT is implemented by a number of prolog files, marked as "provided" in Figure 3.2.

- `trindikit.pl`: Loads the toolkit according to the specifications in the system configuration file (see Section 4.1)
- `tis.pl`: Implements the application of operations and conditions on the TIS; see Section 3.6.
- `datatypes.pl`: Defines general conditions and operations (see Section 3.4); Loads datatypes according to the system configuration file (see Section 4.1).
- `Datatypes/{drs, record, set, stack, stackset, queue, assocSet}.pl`: Provided datatypes[1]. See Section 3.4.
- `Modules/{input_simpletext, interpret_inputform, generate_outputform, output_simpletext}.pl`: Provided modules; see Section 3.5
- `dme_adl.pl`: Interpreter for the DME-ADL language; see Section 3.7.
- `control_adl.pl`: Interpreter for the Control-ADL language; see Section 3.8.
- `flags.pl`: Implementation of system flags and commands for checking and setting them; see Section 3.10.
- `inoutput.pl`: Basic input/output predicates (as for now, only to terminal and to shell scripts); facilities for inspecting TIS na flags, and for printing rules.
- `error.pl`: Error reporting functionality

3.4 Provided datatypes

The TRINDIKIT provides a number of datatypes definitions, to which the user may add his/her own. We currently make a distinction between complex and simple datatypes, where the definitions of the former include conditions and operations on objects of the datatype but the latter does not. Often, the type definition for complex types is inductive while those for simple types are non-inductive.

[1] Not all are completely implemented yet, e.g. the DRS library.

3.4.1 Complex datatypes

Complex datatypes[2] are used extensively in the TRINDI DME architecture, most importantly for modeling the TIS.

The definition of a complex datatype includes of the following:

1. Name
2. Operations
3. Conditions

There should also be a definition of what it means to be an object of the type. This definition can be used by a typechecker.[3]

To make use of a type definition, a *type declaration* is needed which declares a variable to be of a certain type. Resources (including the dynamic information state) are introduced by giving a type definition and declaring a variable to be of that type. When a resource is declared to be of type T, this means that all the conditions and operations available to that type will be available for the resource.

So far, only a limited number of complex datatypes have been defined. Also, the definitions are incomplete in the sense that they don't specify all possible operations and conditions; for example, the operation of forming the union of two sets. Consult the implementation documentation (the README file) for details. Below is a selection of datatypes[4] which are either included in the TRINDIKIT or will be included in future releases. Things which are not implemented in version 1.0 are marked with '*'.

Set

TYPE: SET(T)
DESC: *unordered set of elements of type T*

[2] An alternative term is "datastructure".

[3] Typechecking is not yet fully implemented.

[4] Actually, they are datatype schematas since the type of the included objects is not determined. For example, SET(T) is a datatype schemata but SET(MOVE) is a datatype.

COND:
- **in**(E) : E is unifiable with an element in the set
- **member**(E) : same as **in**(E)
- **strict_member**(E) : E is identical to an element in the set
- **empty** : the set is empty

OP:
- **add**(E) : add E to the set unless it's already a member
- **del**(E) : delete an element unifiable with E from the set
- ***del_all**(E) : delete all elements unifiable with E from the set
- ***del_strict**(E) : delete the element identical with E from the set
- **replace**($E1$,$E2$) : replace an element unifiable with $E1$ with $E2$; equivalent to **del**($E1$) if $E2$ is already in the set
- ***replace_all**($E1$,$E2$) : replace all elements unifiable with $E1$ with $E2$; equivalent to **del_all**($E1$) if $E2$ is already in the set
- ***replace_strict**($E1$,$E2$) : replace the elements identical to $E1$ with $E2$; equivalent to **del_strict**($E1$) if $E2$ is already in the set
- **forall**(El,$NewEl$) : produce a set resulting from replacing all elements unifying with El with $NewEl$
- **extend**(S) : extend the set with another set S

Stack

TYPE: STACK(T)

DESC: *simple stack of elements of type T*

COND:
- **fst**(E) : E is the topmost element on the stack
- **empty** : the stack is empty

OP:
- **push**(E) : pushes E on top of the stack
- **pop** : pops the topmost element off the stack

Stackset

TYPE: STACKSET(T)

DESC: *open stack of elements of type T - allows member check*

COND:
- **fst**(E) : E is the topmost element on the stack
- **in**(E) : E is unifiable with an element in the stack
- **member**(E) : same as **in**(E)
- **strict_member**(E) : E is identical to an element in the stackset
- **empty** : the stack is empty

OP:
- **push(E)** : pushes E on top of the stack
- **pop** : pops the topmost element off the stack
- **del**(E) : delete an element unifiable with E from the set
- ***del_all**(E) : delete all elements unifiable with E from the set
- ***del_strict**(E) : delete the element identical with E from the set

Record

TYPE: RECORD(RT)

DESC: *record of type RT*

COND:
- C#**rec**($Path$, $Arg_1, \ldots, Arg_n$) : $C(Arg_1, \ldots, Arg_n)$ holds of the object which is the value of path $Path$
- **valRecInSetInRec**($Path1$,$Path2$,$Val2$,$Path3$,$Val3$) :

OP:
- O#**rec**($Path$, $Arg_1, \ldots, Arg_n$) : applies $O(Arg_1, \ldots, Arg_n)$ to the object which is the value of path $Path$
- **copyRec**($Path1$,$Path2$) : copies the value of $Path1$ to $Path2$
- **addFieldRec**($Label$,Val) : add a field $Label = Val$ to the record
- **clearAllFieldsRec**(P) :
- **recursiveClearAllFieldsRec**(P) :
- **peRec**($Path_1$,$Path_2$) :
- **peRec**(Rec) :
- **extendRec**($PathA$,$PathB$) :
- **setRecInSetInRec**($Path1$,$Path2$,$Val2$,$Path3$,$Val3$) :
- **setRecinFstRecInStack**($Path1$,$Path2$,Val) :

Queue

TYPE: QUEUE

DESC: *LIFO queue*

COND:
- **empty** : the queue is empty

OP:
- **push**(El) : push El onto the queue
- **last**(El) :
- **extend**(Q) : extend the queue with another queue Q by adjoining the first element of Q with the last element of the queue

Association set

TYPE: ASSOCSET(ElT,AT)

DESC: *association set*[5]

COND:
- **assoc**(El,A) : El is associated with A
- **in**(El,A) : the pair (El,A) is unifiable with an pair in the set
- **empty** : the set is empty

[5] An association set is a set of pairs of type (ElT,AT) where the first element of each pair is said to be *associated* with the second element of the pair.

OP: {
- **add**(El,A) : add the pair (El,A) to the set unless it's already a member
- **del**(El,A) : delete an element unifiable with (El,A) from the set
- **extend**(S) : extend the association set with another association set S

Counter

TYPE: COUNTER
DESC: *counter*
COND: {
OP: {
- **increase** : increase the counter by 1
- **decrease** : decrease the counter by 1

3.4.2 Simple types

For the simple types listed below, type definitions are provided by the toolkit. Type declarations are provided for all these types except MOVE; the designer must provide this declaration, thus specifying the set of dialogue moves available for interpretation, selection and generation. For types which have a small set of objects, these objects are listed.

There are no special conditions or operations for objects of simple types; however, they can be accessed through the general conditions and operations which apply to objects of all types.

TYPE: MOVE
DESC: *dialogue moves*
OBJECTS: { **(user defined)**

TYPE: PARTICIPANT
DESC: *dialogue participant markers*
OBJECTS: {
- **usr**
- **sys**

TYPE: PROGRAM_STATE
DESC: *the state of the system*
OBJECTS: {
- **run**
- **quit**

3.4.3 General conditions and operations

Apart from the type-specific conditions and operations, there are some conditions and operations which apply to all types, including simple types.

TYPE: (ANY TYPE T)
DESC: *conditions and operation which apply to objects of all types*
COND:
- **unify**(*Obj*) : the object unifies with *Obj*
- **equal**(*Obj*) : the object is identical to *Obj*

OP: {

TYPE: (ANY TYPE T)
DESC: *conditions and operation which apply to variables of all types*
COND:
- **unify**(*Obj*) : the value of the variable unifies with *Obj*
- **equal**(*Obj*) : the value of the variable is identical to *Obj*
- **val**(*Obj*) : the value of the variable is *Obj*
- **is_set** : the variable is set (it is not **nil**)
- **is_unset** : the variable is not set (it is **nil**)

OP:
- **set**(*Obj*) : the variable is set to *Obj*
- **unset** : the variable is unset (set to **nil**)
- **clear** : the value of the variable is set to be an empty object of type T

A datatype definition may contain a specification of what the empty object of that type is. If there is no such specification, the empty object is assumed to be **nil**.

3.5 Provided modules

3.5.1 Simple text input module

The input module simply `Modules/input_simpletext.pl` reads a string (until new-line) from the keyboard, preceded by the printing of an input prompt. The system variable input is then set to be the value read.

(1)
```
input :-
        check_condition( input $= _ ),
        flag( input_prompt, Prompt ),
        prompt( OldPrompt, Prompt ),
        read_string( Str ),
        prompt( _, OldPrompt ),
        apply_operation( set( input, Str ) ).
```

3.5.2 Simple text output module

The output module `Modules/output_simpletext.pl` takes the string in the system variable output and prints it on the computer screen, preceeded by the printing of an output prompt. The contents of the output variable is then deleted. The module also moves the contents of the system variable next_moves to the system variable latest_moves. Finally it sets the system variable latest_speaker to be the system.

(2)
```
output :-
          check_condition( is_set( output ) ),
          check_condition( output $= Str ),
          flag( output_prompt, Prompt ),
          name( StrN, Str ),
          write( Prompt ), print( StrN ), nl,
          check_condition( next_moves $= Moves  ),
          apply_operation( unset( next_moves ) ),
          apply_operation( set( latest_moves, Moves ) ),
          apply_operation( set( latest_speaker, sys ) ),
          apply_operation( unset( output ) ).
```

3.5.3 A simple interpretation module

The interpretation module `Modules/interpret_inputform.pl` takes a string of text, turns it into a sequence of words (a "sentence") and produces a set of moves. The "grammar" consists of pairings between lists whose elements are words or semantically constrained variables. Semantic constraints are implemented by a set of semantic categories (`location`, `month`, `task`, `means_of_transport` etc.) and synonomy sets. A synonomy set is a set of word which all are regarded as having the same meaning.

The simplest kind of lexical entry is one without variables. For example, the word "hello" is assumed to realise a `greet` move.:

```
input_form( [hello], greet ).
```

The following rule says that a phrase consisting of the word "to" followed by a phrase S consitutes an `answer` move with content `to=`C provided that the lexical semantics of S is C, and C is a `location`. The lexical semantics of a word is implemented by a coupling between a synset and a meaning; the lexical semantics of S is C, provided that S is a member of a synonomy set of words with the meaning C.

```
input_form([to|S], answer(to=C)):-lexsem(S, C), location(C).
```

To put it simply, the parser tries to divide the sentence into a sequence of phrases (found in the lexicon), covering as many words as possible.

(3)
```
/*-------------------------------------------------------------------
wordlist2moves( +AtomList, -MoveList )
-------------------------------------------------------------------*/

wordlist2moves( [], [] ).
wordlist2moves( Sentence, [Move|Moves] ) :-
        append( Phrase, Rest, Sentence ),
        condition( lexicon: input_form(Phrase,Move) ),
        wordlist2moves( Rest, Moves ).
wordlist2moves( [Word|Sentence], Moves ) :-
        \+ condition( lexicon: input_form([Word],_) ),
        wordlist2moves( Sentence, Moves ).
```

First, if the sentence is empty, there are no moves. Second, if Sentence begins with a Phrase, for which there is some Move definied in the lexicon, then it also tries to find the Moves for the Rest of the Sentence. Third, if the first Word is not defined as a phrase in the lexicon, it just skips that Word.

3.5.4 A simple generation module

The generation module `Modules/generate_outputform.pl` takes a sequence (list) of moves and outputs a string. The generation grammar/lexicon is a list of pairs of move templates and strings.

```
output_form( greet, "Welcome to the travel agency!" ).
```

In some cases, the move template contains some variable which is assumed to be instantiated when the lexicon is consulted. The lexicon will then find a string corresponding to the instantiated variable and insert it into the output string.

(4)
```
output_form( answer(X^(price=X),price=Price), Str ) :-
        number( Price ), number_chars( Price, PriceStr ),
        append( "It will cost ", PriceStr, Str1 ),
        append( Str1, " crowns", Str ).
```

To realize a list of Moves, the generator looks, for each move, in the lexicon for the corresponding output form (as a string), and then appends all these strings together. The output strings is appended in the same order as the moves.

3.6 Accessing the Total Information State

The TIS is read using the predicates `check_conditions/1`, whose argument is a list of conditions on the TIS, or `check_condition/1` whose argument is a single condition. It is written to using `apply_operations/1`, whose argument is a list of operations on the TIS, or `apply_operation/1`. In the following two sections, the syntax of TIS conditions and operations is described.

3.6.1 TIS condition syntax

Var is a TIS variable. $Term$ is a prolog term.

- C `(` Var `,` Arg_1 `,` ... `,` Arg_n `)`
  There is a condition C, taking arguments $Args_1$, ... $Args_n$, defined for the type that Var is of.
  Example: `in(latest_moves,answer(dest(london)))` – uses the condition `in/1` defined for e.g. sets[6]

- C `(` Arg_1 `,` ... `,` Arg_n `)`
  Same as C `(` `is` `,` Arg_1 `,` ... `,` Arg_n `)`. This is a shortcut for writing conditions on the IS variable[7].
  Example: `push#rec(shared.qud,Q)` – equivalent to `push#rec(is,shared.qud,Q)`

- `!` Cond
  Cond holds; if not, report error
  Example: `! in(latest_moves,answer(dest(london)))`

- $Cond_1$ `and` $Cond_2$
  $Cond_1$ and $Cond_2$ both hold
  Example: `in(Moves, answer(A)) and lexicon::yn_answer(A)`

- $Cond_1$ `or` $Cond_2$
  $Cond_1$ or $Cond_2$ holds
  Example: `(Move=answer(_,_)) or (Move=thank)`

- `not` $Cond_1$
  $Cond_1$ does not hold
  Example: `not in#rec(private.plan, respond(Q))`

[6] This condition should more properly be called `contains` due to the order of its arguments.

[7] To resolve possible ambiguities in the meaning of conditions caused by this shortcut, the condition interpreter checks the number of arguments of the condition to see if the shortcut interpretation applies. This requires that the condition does not have two separate definitions with differing arity of one argument.

- $Term_1$ `=` $Term_2$
  $Term_1$ and $Term_2$ unify
  Example: `Move=ask(_)`

- $Term_1$ `\=` $Term_2$
  $Term_1$ and $Term_2$ do not unify

- $Term_1$ `==` $Term_2$
  $Term_1$ and $Term_2$ are identical

- $Term_1$ `\==` $Term_2$
  $Term_1$ and $Term_2$ are not identical
  Example: `OldTask \==NewTask`

- Var `$=` $Term$
  The value of Var unifies with $Term$
  Example: `latest_moves$=Moves`

- Var `$==` $Term$
  The value of Var is identical to $Term$
  Example: `latest_speaker$=sys`

- `result(` O `(` $Source$ `,` $Arg_1 \ldots Arg_n$ `)` `,` $Result$ `)`
  $Result$ is the result of applying the operation O to $Source$, with arguments $Arg_1 \ldots Arg_n$. In conditions, Result must be a prolog variable which gets instantiated.
  Example: `result(add(MoveSet1,Move),MoveSet2)` – `MoveSet2` is the result of adding `Move` to `MoveSet1`

- $Macro$
  The list of conditions corresponding macros $Macro$ which all hold
  Example: `setIntegrateFlag(ask(Q), true)`

- C`#`$Type$ `(` Var, $Path$, $Arg_1 \ldots Arg_n$ `)`
  C `(` Var, $Arg_1 \ldots Arg_n$ `)` holds for an object embedded in an object of type $Type$ at $Path$ [8].
  Example: `in#rec(shared.bel, A)`

- Var `::` C `(` $Arg_1 \ldots Arg_n$ `)`
  same as C `(` Var `,` $Arg_1 \ldots Arg_n$ `)`
  Example: `latest_moves :: in(answer(dest(london)))`

- `result(` $Source$ `,` $Result$ `)` `::` O `(` $Arg_1 \ldots Arg_n$ `)`
  Same as `result(` O `(` $Source$ `,` $Arg_1 \ldots Arg_n$ `)` `,` $Result$ `)`
  Example: `result(MoveSet1,MoveSet2)::add(Move)`

[8] Only works for some datatypes, e.g. records.

3.6.2 TIS Operation syntax

Source is either a TIS variable or a prolog term. *Store* is either a TIS variable or a prolog variable.

- O (*Source* , Arg_1 , ... , Arg_n)
  There is an operation O, taking arguments $Args_1$, ... $Args_n$, defined for the type that *Source* is of. The result of applying the operation to *Source* is stored in *Source* itself.
  Example: `add(latest_moves,answer(dest(london)))` – uses the condition `add/1` defined for e.g. sets

- O (Arg_1 , ... , Arg_n)
  Same as O (`is` , Arg_1 , ... , Arg_n). This is a shortcut for writing conditions on the IS variable[9].
  Example: `push#rec(shared.qud,Q)` – equivalent to `push#rec(is,shared.qud,Q)`

- `result(` O (*Source* , $Arg_1 \ldots Arg_n$) , *Store*)
  There is an operation O defined for the type that both *Source* and *Store* are of. Store the result of applying the operation O to *Source*, with arguments $Arg_1 \ldots Arg_n$, in *Store*. If *Store* is a TIS variable, it gets set to the result; if *Store* is a prolog variable, it becomes instantiated with the result.
  Example: `result(add(latest_moves,Move),MoveSet2)` – `MoveSet2` is the result of adding `Move` to the value of `latest_moves`

- ! *Cond*
  Cond holds; if not, report error
  Example: `! ( database :: consultDB( Q, SharedBel, R ) )`

- *Macro*
  Apply the list of operations corresponding to macro *Macro*
  Example: `setIntegrateFlag(ask(Q), true)`

- `create_empty(`*PseudoType*`,` *Var*`)`
  Var is instantiated with an empty object of type *PseudoType*, where *PseudoType* is e.g. SET rather than SET(MOVE)
  Example: `create_empty( record, MRec0 )` – instantiates `MRec0` to `record([])`

- *Source* :: O ($Arg_1 \ldots Arg_n$)
  same as O (*Source* , $Arg_1 \ldots Arg_n$)
  Example: `latest_moves :: add(answer(dest(london)))`

[9] To resolve possible ambiguities in the meaning of operations caused by this shortcut, the operation interpreter checks the number of arguments of the condition to see if the shortcut interpretation applies. This requires that the operation does not have two separate definitions with differing arity of one argument.

- `result(` *Source* `,` *Store* `) :: ` *O* `(` $Arg_1 \ldots Arg_n$ `)`
  Same as `result(` *O* `(` *Source* `,` $Arg_1 \ldots Arg_n$ `) ,` *Store* `)`
  Example: `result(latest_moves,MoveSet2)::add(Move)`

3.7 The DME-ADL language

DME-ADL (Dialogue Move Engine Algorithm Definition Language) is a language for writing algorithms for updating the TIS. Algorithms in DME-ADL are expressions of any of the following kinds (C is a TIS condition; R, S and T are algorithms, and RC is a rule class):

1. RC
   apply an update rule of class RC; rules are tried in order
2. `[`$R_1, \ldots, R_n$`]`
   execute $R_1, \ldots, R_n$ in sequence
3. `if` C `then` S `else` T
   If C is true of the TIS, execute S; otherwise, execute T
   Example: `if (latest_speaker $== usr) then ([ (repeat refill), (try database) ]) else store`
4. `while` C `do` R
   while C is true of the TIS, execute R repeatedly
5. `repeat` R `until` C
   execute R repeatedly until C is true of the TIS
6. `repeat` R
   execute R repeatedly until it fails; report no error when it fails
   Example: `repeat refill`
7. `repeat+` R
   execute R repeatedly, but at least once, until it fails; report no error when it fails
8. `try` R
   try to execute R; if it fails, report no error
9. R `or` S
   Try to execute R; if it fails, report no error and execute S instead
10. `test` C
    if C is true of the TIS, do nothing; otherwise, halt execution of the current algorithm

11. *SubAlg*
    execute subalgorithm *SubAlg*

Subalgorithms are declared using `==>`, which is preceded by the subalgorithm name and followed by the algorithm, as in (5).

(5)
```
main_update ==> [ grounding, repeat+ ( integrate or
accommodate ) ].
```

A sample DME-ADL algorithm is shown in (6).

(6)
```
if (latest_moves $== failed)
  then (repeat refill)
  else (   ! [ grounding,
               repeat+ ( integrate or accommodate  ),
               (   if (latest_speaker $== usr)
                      then ([ (repeat refill),
                              (try database) ])
                      else store
               )
             ] ).
```

3.8 The Control-ADL language

The Control-ADL language is identical to the DME-ADL language, except that it calls modules instead of rule classes, and it can include the `print_state` instruction. Modules are called by a 0-ary predicate associated with the module, as specified by the `selected_modules` predicate in the system configuration file (see Section 4.1).

The syntax for the Control-ADL language is the same as for DME-ADL, except for the first expression, which is replace by that in (7), and the `print_state` expression described in (8).

(7) *Module*
    Call *ModulePred*, where *ModulePred* is a 0-ary predicate for running a module

(8) `print_state`
    Print the TIS or IS, provided the value of the `show_state` is `all` or `is`, respectively

A sample Control-ADL algorithm is shown in (9).

(9)
```
[reset,
 repeat ( [ select,
            generate ,
            output,
            update,
            print_state,
            test( program_state $== run ),
            input,
            interpret,
            print_state,
            update,
            print_state
          ] )
]
```

3.9 Inspecting the TIS

There are a couple of predicates for inspecting the TIS from the prolog shell (the `user` prolog module).

- `tis/0`: prints the TIS
- `is/0`: prints the IS
- `ivs/0`: prints the module interface variables
- `rvs/0`: prints the resource interface variables

Also, TIS variables can be set using `set(+Var,+Val)` and their value can be checked using `val(+Var,-Val)`.

3.10 Inspecting flags

To change the value of a flag, use `setflag(+Flag,+Value)`, as in (10).

(10)
```
setflag(access_restrictions,off).
```

The predicate `flags/0` prints all flags, their current values and a short description of each flag.

```
Flag                 Possible Values          Current Value  Description
output_prompt         [\n$S> ,\n$U> ]         \n$S>            The output prompt
input_prompt          [\n$S> ,\n$U> ]         \n$U>            The input prompt
language             [english,swedish]        english        Language
domain              [travel,autoroute]        travel         Domain
typecheck                [yes,no]              no             Type checking
format            [text,shell,latex,html]     text           The output format
show_state               [all,is,no]          all            Show the information state variables
show_rules                [yes,no]            yes            Show the update/select rules
output_stream              [user]             user           The output stream
version              [web,standalone]         standalone     Demo type
access_restrictions       [on,off]            on             Restrictions on module access to TIS
```

Chapter 4

How to implement a dialogue system using the TRINDIKIT

In this chapter, we will try to show how one goes about building a dialogue system using the TRINDIKIT. Of course, building a dialogue system requires a lot more than knowing how to use the TRINDIKIT; most importantly, it requires some theory of dialogue moves, information states and information state updates. Most examples in this chapter are from the GoDiS system. A template system `mySystem` is included in the TRINDIKIT package.

4.1 The system configuration file

The system configuration file specifies how the system is constructed. It specifies what datatypes, modules, resources and macros are used, and defines the the predicate which runs the system. It may also include flag settings.

In this section, we will review the contents of the configuration file. For a sample configuration file, see `GoDiS/godis.pl`. A template configuration file `mySystem/ mySystem.pl` is also available.

4.1.1 Selecting datatypes and macros

Datatypes are selected using `selected_datatypes/1`, where the argument is a list of datatypes to be loaded. Each item DataType in the list corresponds to a file DataType.pl

in the search path.

(11) (in system configuration file:)

```
selected_datatypes([standard, record, set, stack,
stackset, assocSet, godis_datatypes]).
```

Optionally, macros may be used to access the TIS. Macros are selected by defining the predicate `selected_macro_files/1`, whose argument is a list of files in the search path which contain macro definitions.

(12) (in system configuration file:)

```
selected_macro_file( godis_macros ).
```

4.1.2 Selecting modules

Modules are selected using `selected_modules/1`, where the argument is a list of TRINDIKIT modules to be loaded. Each module spec has the form *Predicate*:*FileName*, where Predicate is the 0-ary predicate used to call the module, and FileName.pl is the a file in the search path containing the module specification.

(13) (in system configuration file:)

```
selected_modules([ control: control,
                   input : input_simpletext,
                   interpret : interpret_simple1,
                   update : update,
                   select : select,
                   generate : generate_simple1,
                   output : output_simpletext,
                   reset : reset
                 ]).
```

4.1.3 Specifying the DME

The DME modules are specified using `dme_modules/1`, whose argument is the list of DME modules. These modules have unlimited acccess to the TIS.

(14) (in system configuration file:)

```
dme_modules([ update, select ]).
```

4.1.4 Loading and selecting resources

The predicate `load_resources/1` is called to load the resources that should available for hookup to the TIS. Its argument is a list of resource filenames, as in (15),

(15) (in system configuration file:)

```
:- load_resources([lexicon_travel_english,
lexicon_travel_swedish, lexicon_autoroue_english, domain_travel,
domain_autoroute, database_travel, database_autoroute]).
```

The resources are selected by setting the resource variables to the appropriate values. Each resource R corresponds to a file F`.pl` which defines a prolog module F. The only difference between R and F is that each "-" in R is replaced by a "_" in F[1].

For example, assume we have three resource variables lexicon, database and domain, and three resources `lexicon_travel_english`, `database_travel` and `domain_travel`, respectively. The predicate defined in (16) will select these resources by setting the resource variables to the appropriate values.

(16) (in system configuration file:)

```
set_resource_variables:-
        set( lexicon, lexicon-travel-english ),
        set( database, database-travel ),
        set( domain, domain-travel ).
```

The resource interface variables are part of the TIS and can thus change values during system runtime.

4.1.5 File search paths

It must also specify the file search path where the TRINDIKIT and the system itself is found. Currently, this is done using using the `assertz/1` predicate.

[1] This is done for easier processing of the resource filename in prolog.

(17) (in system configuration file:)

```
:- assertz(user:file_search_path(home, '$HOME')).
:- assertz(user:library_directory(home('<trindikitpath>'))).
:- assertz(user:library_directory(home('<trindikitpath>/Datatypes'))).
:- assertz(user:library_directory(home('<trindikitpath>/Modules'))).
:- assertz(user:library_directory(home('<systempath>'))).
:- assertz(user:library_directory(home('<systempath>/Datatypes'))).
:- assertz(user:library_directory(home('<systempath>/Modules'))).
```

4.1.6 Loading the TRINDIKIT

The configuration file must also contain the line in (18). This line must come after the abovementioned definitions.

(18) (in system configuration file:)

```
:- ensure_loaded( library(trindikit) ).
```

4.1.7 The run predicate

In its simplest form, the run predicate simply calls the control module. This requires that all resources variables have been set. However, one may want to set variables and flags in runtime before calling the control module.

4.2 Specifying Total Information State

The Total Information State (TIS) consists of the Information State (IS) variable, module interface variables and resource interface variables. It is accessed using conditions and operations. The TIS is specified by giving type declarations for the TIS variables, thereby specifying which conditions and operations are available.

Type declarations state that a TIS variable has a certain type. Objects are possible values of a variable. If a variable has type T it can take as values objects of type T. The fact that an object has a certain type is specified using `of_type/2`.

4.2.1 Specifying information state type

The user-defined component `is_type`, implemented in the file `is_type.pl` exports the predicate `infostate_variable_of_type/2` to the TIS. This predicate declares the type of the IS variable.

(19) (in `is_type.pl`:)

```
:- module(is_type, [infostate_variable_of_type/2]).

infostate_variable_of_type( is, IsType ) :-
        IsType = record( [ private:Private,
                           shared:Shared ] ),
        LM = record( [ speaker:speaker,
                       moves:assocSet( dmoverec ) ] ),
        Shared = record( [ bel:set( proposition ),
                           qud:stackset( question ),
                           lm:LM ] ),
        Private = record( [ agenda:stack( action ),
                            plan:stackset( action ),
                            bel:set( proposition ),
                            tmp:Shared ] ).
```

It is in principle possible to have more than one infostate variable. If there is only one and its name is `is`, it is possible to use the shorthand format for conditions and operations, where the `is` can be left out.

4.2.2 Specifying module interface variables

The user-defined component `module_interfaces`, implemented in the file `module_interfaces.pl` exports the predicate `interface_variable_of_type/2` to the TIS. This predicate declares the type of the interface variables.

(20) (in `module_interfaces.pl`:)

```
:- module( interface_variables, [ interface_variable_of_type/2 ] ).

interface_variable_of_type( input, string ).
interface_variable_of_type( output, string ).
interface_variable_of_type( latest_speaker, speaker ).
interface_variable_of_type( latest_moves, set(move) ).
interface_variable_of_type( next_moves, set(move) ).
interface_variable_of_type( program_state, program_state ).
```

4.2.3 Resource interface definitions

The user-defined component `resource_interfaces`, implemented in the file `resource_interfaces.pl` exports the predicate `interface_variable_of_type/2` to the TIS. This predicate declares the type of the resource variables. It also exports the predicates `resource_condition/3`, `resource_operation/4` and `is_resource_type/2` to the `datatypes` component.

Each resource is accessed through a variable of a datatype with associated conditions and (in some cases) operations. There may be several objects of each type, corresponding to instantiations of the resource. As an example, the interface definition for the GoDiS lexicon is shown in (21).

(21) (in `resource_interfaces.pl`:)

```
:- module(resource_interfaces, [
                                 resource_variable_of_type/2,
                                 is_resource_type/1,
                                 resource_condition/2,
                                 resource_operation/2
                                ]).

is_resource_type( lexicon ).

of_type( lexicon-travel-english, lexicon ).
of_type( lexicon-autoroute-english, lexicon ).
of_type( lexicon-travel-swedish, lexicon ).

resource_variable_of_type( lexicon, lexicon ).

resource_condition( lexicon, input_form( Phrase, Move ), Lexicon ) :-
        Lexicon : input_form( Phrase, Move ).

resource_condition( lexicon, output_form( Phrase, Move ), Lexicon ) :-
        Lexicon : output_form( Phrase, Move ).

resource_condition( lexicon, yn_answer(A), Lexicon ) :-
        Lexicon : yn_answer( A ).
```

Here, the `lexicon` variable has type `lexicon` (perhaps somewhat confusing) and there are two objects of the `lexicon` type: `travel-swedish` and `travel-english`, corresponding to the swedish and english travel domain lexicons, respectively. To select a lexicon, the `lexicon` variable is set to the appropriate value.

Resource conditions and operations check the current value of the interface variable in question, and passes the condition/operation on the selected resource. This enables dy-

namic switching between resources during dialogue, e.g. to move to a new domain or change language.

Technically, this is implemented using prolog modules. Each resource object is a module (stored in a file with the same name as the module, which is also the name of the object). Depending on the value of the resource variable, the resource interface inspects different prolog modules, i.e. different resource objects.

4.2.4 Macros

Macros are defined by associating a macro with a list of TIS conditions or a list of TIS operations. To specify a condition macro, `macro_cond/2` is used; for operation macros use `macro_op/2`. A sample precondition macro is shown in (22).

(22) (in macro file:)

```
macro_cond( movesInRec( Path, Moves ),
               [valRec( Path, MoveRecs ),
                result(
                   forall( MoveRecs, record([move=Move|_]), Move ),
                     Moves )
            ] ).
```

To specify which file contains the macros, use selected_macro_file/1, as in (23).

(23) (in system configuration file:)

```
selected_macro_file( godis_macros ).
```

4.3 Building DME modules

A DME module consists of a set of rules and and algorithm. The algorithm can either use DME-ADL or be written in prolog. If it uses the DME-ADL, the interpreter must be imported into the DME module. DME modules have unlimited access to the TIS.

A DME module consists of the following parts:

- a module declaration

- importation of predicates from other modules
- loading of rules
- (optionally) loading of the DME-ADL interpreter
- the DME algorithm itself
- the module call predicate

4.3.1 Module Declaration

The module is declared as a prolog module which exports the module call predicate (see Section 4.3.6).

(24)
```
:- module(update,[update/0]).
```

4.3.2 Importing predicates from other modules

Usually, at least the predicates in (25) must be imported into the control module.

(25)
```
:- use_module(library(tis),[check_condition/1,check_conditions/1,
                apply_operations/1, apply_operation/1]).
:- use_module(library(inoutput),[print_rule/1]).
:- use_module(library(error), [error/1]).
```

4.3.3 Load rules

The rules are loaded using `:- ensure_loaded(library(`*RuleFile*`))`, where *RuleFile* is the file containing the rule definitions for the module in question. An example is seen in (26).

(26)
```
:- ensure_loaded(library(update_rules)).
```

4.3.4 Load the DME-ADL interpreter

(27)
```
:- ensure_loaded(library(dme_adl)).
```

4.3.5 The update algorithm

A sample algorithm which uses DME-ADL is shown in (28).

(28)
```
update_algorithm(
                    if (latest_moves $== failed)
                    then (repeat refill)
                    else
                   (   ! [ grounding,
                           repeat+ ( integrate or accommodate  ),
                           (   if (latest_speaker $== usr)
                               then ([ (repeat refill),
                                       (try database) ])
                               else store
                           )
                         ] ) ).
```

4.3.6 Module call predicate

The module call predicate is simple the predicate used to call the module. It is a 0-ary predicate which is exported by the module. To use the DME-ADL interpreter, pass the specified algorithm as the argument to `adl_exec/1`.

(29)
```
update :-
        update_algorithm( Algorithm ),
        adl_exec( Algorithm ).
```

4.3.7 Writing update rules

Each DME module requires a file (prolog module) with update rules. This file should define a prolog module which exports the predicates `rule/3` and `rule_class/2`. To be able to handle the TIS condition and operation syntax, it also needs to define some operators as seen in (30).

(30)
```
:- module(update_rules, [rule/3, rule_class/2]).

:- op(800, fx, ['!', not]).
:- op(850, xfx, ['$=', '$==', and, or] ).
```

The syntax for update rule definitions is shown in (31).

(31)
```
rule( RuleName, PrecondList, EffectsList )    of_class(
RuleName, RuleClass )
```

Here, `PrecondList` is a list of TIS conditions (Section 3.6.1) and `EffectsList` is a list of TIS operations (Section 3.6.2). The `RuleClass` may be used in defining DME algorithms (Section 3.7.

A sample update rule definition is shown in (32).

(32)
```
urule( integrateUserQuestion,
       integrate,
       [ val#rec( shared^lm^speaker, usr ),
         moveInRec( shared^lm^moves,  ask(Q) ),
         valIntegrateFlag( ask(Q), false ),
         not in#rec( private^plan, respond(Q) )
       ],
       [ push#rec( shared^qud, Q ),
         push#rec( private^agenda, respond(Q) ),
         setIntegrateFlag( ask(Q), true )
       ]
     ).

of\_class( integrateUserQuestion, integrate )
```

4.4 Building a Control module

The control module determines the order in which the other modules are called[2]. It can inspect specified parts of the TIS. The control module must include specifications of access restrictions, as for other non-DME modules (see Section 4.5.1).

A control module consists of the following parts:

- a module declaration
- importation of predicates from other modules
- specification of TIS access restrictions
- (optionally) loading of the Control-ADL interpreter
- the control algorithm itself

[2]In the current version, it is not possible to run modules asynchronously; however, we intend to include this facility in a future version.

4.4.1 Module declaration

(33)
```
:- module(control,[control/0]).
```

4.4.2 Importing predicates from other modules

Usually, at least the predicates in (34) must be imported into the control module.

(34)
```
:- use_module(library(flags), [flag/2]).
:- use_module(library(tis),[check_condition/1]).
:- use_module(library(error),[error/1]).
:- use_module(library(inoutput),[print_state/0]).
```

4.4.3 TIS access restrictions

The control module has limited access to the TIS. The arguments of the predicates `read_access/1` and `write_access/1` are lists of TIS variables that the module is allowed to read from and write to, respectively.

For example, if the input module contains the lines in nexteg, this module can only read the `program_state` interface variable.

(35)
```
read_access( [ program_state ] ).
write_access( [] ).
```

4.4.4 Load the Control-ADL interpreter

(36)
```
:- ensure_loaded( library( control_adl ) ).
```

4.4.5 The control algorithm

A sample control algorithm which uses Control-ADL is shown in (37).

(37)
```
control_algorithm( [reset,
                    repeat (  [ select,
                                generate ,
                                output,
                                update,
                                print_state,
                                test( program_state $== run ),
                                input,
                                interpret,
                                update,
                                print_state
                              ] )
                   ] ).
```

If Control-ADL is not used, modules are called using the appropriate predicates as specified in the system configuration file. Conditions on the TIS are implemented using the `check_condition/1` predicate, whose argument is a TIS condition. The algorithm in (38) works the same way as that in (37), but does not utilize Control-ADL.

(38)
```
control :-
        reset,
        select,
        generate,
        output,
        update,
        print_state,
        control_loop.

control_loop :-
        check_condition( program_state $== run ),
        input,
        interpret,
        update,
        print_state,
        select,
        generate,
        output,
        update,
        print_state,
        control_loop.

control_loop :-
        val( program_state, quit ).
```

4.5 Building non-DME modules

Non-DME modules can either use the DME-ADL interpreter, or be written in plain prolog[3]. In the former case, the non-DME module is built the same way a DME module is, apart for the fact that the non-DME modules must specify TIS access restrictions (see below, Section 4.5.1). In the latter case, no special restrictions apply (apart from the access restriction specification). Conditions on the TIS are checked using the `check_condition/1` predicate, whose argument is a TIS condition. Operations on the TIS are applied using `apply_operation/1`, whose argument is a TIS operation.

4.5.1 Access restrictions for non-DME modules

Non-DME modules have limited access to the TIS, and should preferably only be allowed to read and write to dedicated interface variables. The arguments of the predicates `read_access/1` and `write_access/1` are lists of TIS variables that the module is allowed to read from and write to, respectively.

For example, if the input module contains the lines in (39), this module can read and write only to the input variable.

(39)
```
write_access( [input] ).
read_access( [input] ).
```

4.6 Connecting resources to the TIS

Resources must be connected to the TIS via a resource interface definition (see Section 4.2.3). That is, the resource interface (a prolog module) must export the predicates needed for giving a type declaration of the corresponding resource interface variable.

The file which defines the resource must include a module declaration, as in (40). Note that no predicates need to be exported; the resource interface accesses the resource by "peeking" inside the resource module.

(40)
```
:- module( lexicon_travel_english ).
```

[3]The current TRINIDKIT implementation supports prolog only; future versions may support other languages as well.

4.7 Adding new datatypes

Definitions of new datatypes can be implemented in any file or number of files. To use a datatype, include the filename in the list specified by the `selected_datatypes/1` predicate in the system configuration file (see Section 4.1). The TRINDIKIT will load the definitions into the `datatypes` prolog module.

4.7.1 Complex datatypes

To define an complex datatype T one needs to declare

- That T is a type, using `is_type/1`. This declaration can be conditional on the type of the elements in the complex type, e.g. `is_type(set(Type)) :- is_type(Type)`.
- What it means to be of type T, i.e. the requirements on objects of type T, using `of_type/1`. This definition may be inductive, i.e. the type of a complex type may depend on the types of its embedded objects (see (41) for an example).
- What an empty object of the type is, using `empty_object/1`
- Conditions and operations on objects of type T, using `condition/3` and `operation/4`

As an example, the TRINDIKIT definition of a stack is shown in (41). Note that the predicates used to define a datatype must be declared to be `multifile`.

(41)
```
:- multifile is_type/1, of_type/2, empty_object/2, operation/4, condition/3.

is_type( stack(Type) ) :- is_type( Type ).

of_type( stack([]), stack(_) ).
of_type( stack([Object|Stack]), stack(Type) ) :-
        of_type( Object, Type ),
        of_type( stack(Stack), stack(Type) ).

empty_object( stack(_), stack([]) ).

condition( stack(_), empty, stack([]) ).
condition( stack(_), fst(Fst), stack([Fst|_]) ).

operation( stack(_), push(Fst), stack(Stack), stack([Fst|Stack]) ).
operation( stack(_), pop, stack([_|Stack]), stack(Stack) ).
operation( stack(_), extend(StackA), StackB, StackBA ):-
        StackA = stack(A),
        StackB = stack(B),
        append( B, A, BA ),
        StackBA = stack(BA).
```

4.7.2 Simple types

Simple types are defined using `is_type/1` and the objects are defined using `of_type/2`.

(42)
```
is_type( action ).

of_type( quit, action ).
of_type( greet, action ).
of_type( respond(Q), action ) :-
        of_type( Q, question ).
of_type( raise(Q), action ) :-
        of_type( Q, question ).
```

4.8 User flags

User flag definitions should be placed in a file called `user_flags.pl`, which will be automatically loaded by the TRINDIKIT. User flags are specified using two predicates: `flagValue/2` and `flagInfo/3`, which both must be declared to be `multifile` and `dynamic`. The former assigns default values to the flags, and the latter associates each flag to a list of possible values and a free-text description of the flag. As an example, the flags specified by GoDiS are shown in (43).

(43) (in user_flags.pl:)

```
:- multifile flagValue/2, flagInfo/3.
:- dynamic flagValue/2, flagInfo/3.

flagValue( output_prompt, '\n$S> ' ).
flagValue( input_prompt, '\n$U> ' ).
flagValue( language, english ).
flagValue( domain, travel ).

flagInfo( output_prompt, ['\\n$S> ','\\n$U> '], 'The output prompt' ).
flagInfo( input_prompt, ['\\n$S> ','\\n$U> '], 'The input prompt' ).
flagInfo( language, [ english, swedish], 'Language' ).
flagInfo( domain, [ travel, autoroute ], 'Domain' ).
```

Bibliography

Bos, J., Bohlin, P., Larsson, S., Lewin, I., and Matheson, C. (1999). Dialogue dynamics in restricted dialogue systems. Technical Report Deliverable D3.2, Trindi.

Traum, D., Bos, J., Cooper, R., Larsson, S., Lewin, I., Matheson, C., and Poesio, M. (1999). A model of dialogue moves and information state revision. Technical Report Deliverable D2.1, Trindi.