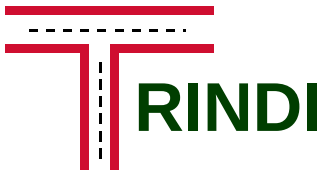
# TrindiKit 2.0 Manual
# - revised version -

Staffan Larsson Alexander Berman Johan Bos
Leif Grönqvist Peter Ljunglöf David Traum

Distribution: PUBLIC




Task Oriented Instructional Dialogue
LE4-8314

Deliverable D5.3 – Manual

October 30, 2000

LE4-8314 TRINDI

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.

Responsibility for authorship is divided as follows. Staffan Larsson was the overall editor and wrote chapters 1, 2, 3 and 4 and appendix A, except section 2.7.3 which was written by Alexander Bermann. Appendix B was written by Staffan Larsson and Peter Ljunglöf. Appendix C was written by Alexander Berman and Staffan Larsson. Appendix D was written by Leif Grönqvist. Johan Bos and David Traum provided comments and suggestions.

The TrindiKit architecture was a collaborative result of the TRINDI Consortium. TrindiKit 2.0 was implemented by Staffan Larsson, Alexander Berman, Peter Ljunglöf, Johan Bos and Leif Grönqvist, with suggestions and advice from Robin Cooper, Torbjörn Lager, Ian Lewin and David Traum.

# 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*, or *DME*, 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. As from version 2.0, TrindiKit supports asynchronous (concurrent) control [1]. 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).

[1] In this document, we will use the terms "asynchronous" and "concurrent" interchangeably, to designate systems where several algorithms can be executed independently as separate processes. We use the terms "serial" or "synchronous" to designate systems which are not asynchronous. Note that we regard this distinction as independent of how data flows in a system; for example, so-called *pipeline* systems can be implemented either asynchronously or serially.

Note that TRINDI deliverable D5.3 is the actual TrindiKit implementation[2], 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 versions of the TrindiKit and manual are available from the TRINDI web page.

## 1.1 The TrindiKit architecture

This toolkit implements an architecture based on the notion of an information state. A system consists of a number of modules (including speech recognizer and synthesizer, natural language interpretation and generation, and a Dialogue Move Engine) which can read from and write to the information state using information state update rules. External resources can be hooked up to the information state. A controller wires the modules together.

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.

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

The standard components of the architecture are the

- the Total Information State (TIS), consisting of
  - the Information State (IS) variable
  - module interface variables
  - resource interface variables
- modules, operating according to module algorithms
- the Dialogue Move Engine (DME), consisting of one or more modules; the DME is responsible for updating the IS based on observed moves, and selecting moves to be performed by the system.
- a controller, wiring together the other modules, either in sequence or through some asynchronous mechanism.
- resources, such as databases etc.

[2]The current TrindiKit implementation is written in SICStus Prolog.

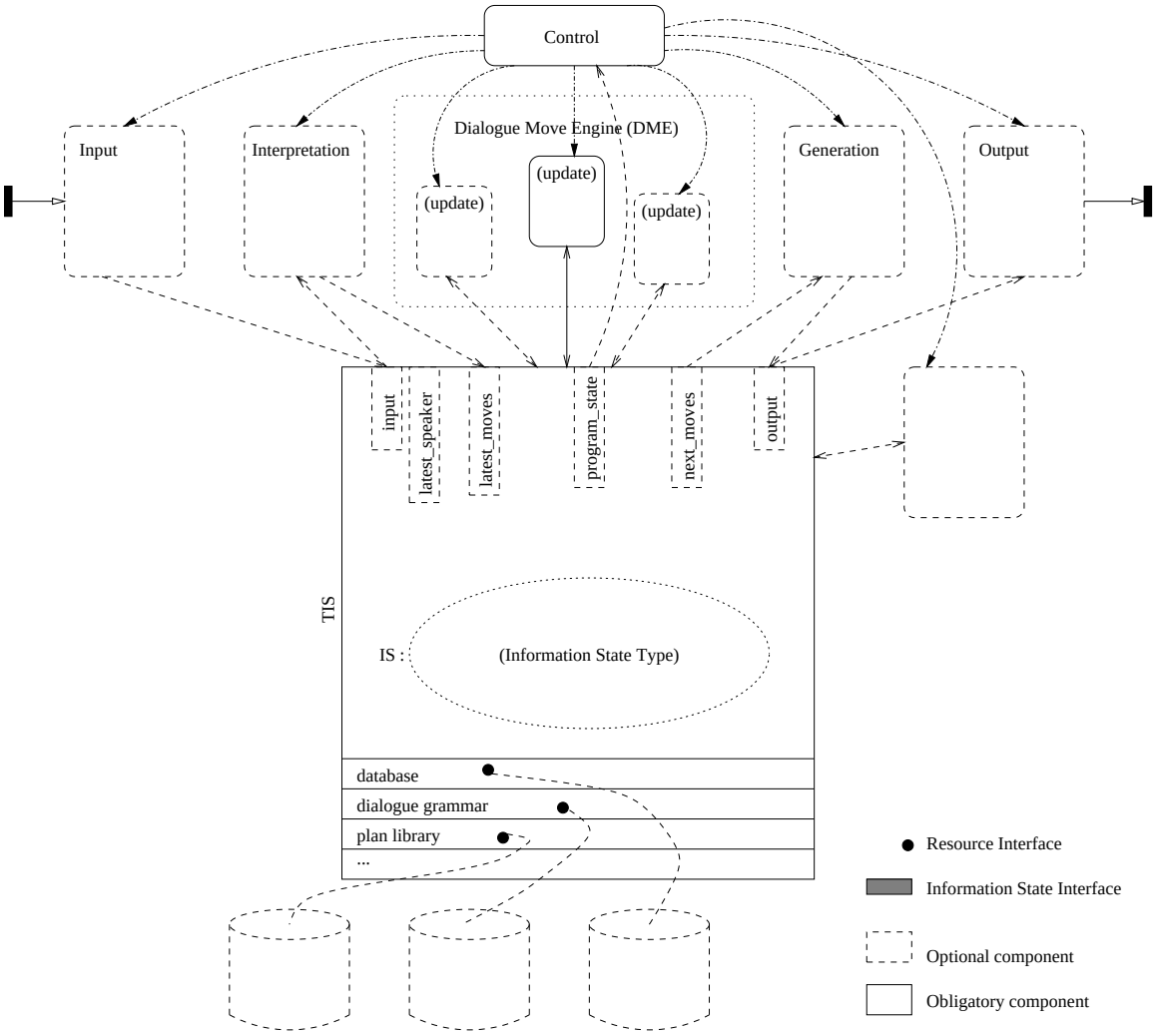



Figure 1.1: Sketch of a sample setup of a system using the TrindiKit architecture

The TIS is accessed by modules through conditions and operations. The types of the various components of the TIS determine which conditions and operations are available.

Some of the components are obligatory (i.e. all systems built using the TrindiKit must have them), and others are optional. To build a system, one must minimally supply

- An Information State type declaration.
- At least one module, consisting of TIS update rules and a module algorithm
- A controller, 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 module 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 System requirements

TrindiKit 2.0 requires SICStus Prolog[3] version 3.8[4]. Serial systems can run on any platform where SICStus Prolog is installed. Asynchronous systems run on UNIX and Linux platforms only, and require the AE package supplied with the TrindiKit, or OAA version 2.12 or later (available from `www.ai.sri.com/~oaa/`).

The latest distribution of TrindiKit can be downloaded from the TrindiKit homepage, `www.ling.gu.se/research/projects/trindi/trindikit/`.

[3] `http://www.sics.se/sicstus`

[4] Version 3.7 may also work; older versions are not recommended.

# Chapter 2

# TrindiKit Concepts

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

## 2.1 Total Information State

The Total Information State (TIS) consists of three components: the information state variable (IS), the module interface variables (MIVs), and the resource interface variables (RIVs). These variables go under the collective name *TIS variables*.

## 2.2 Information State

The information state ("the information state *proper*") 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 (checked, queried and updated) by modules

## 2.3 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.

## 2.4 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.

## 2.5 Modules

Modules[1] have the following basic characteristics:

- they execute update algorithms
- they can read and/or write to the total information state
- they are called by the controller, and can be run serially or asynchronously with other modules
- they can *not* be called from update rules

Every module specifies one or more algorithms. The default is that each module specifies a single algorithm which has the same name as the module itself.

[1] In this document, the term "module" refers to TrindiKit modules rather than e.g. Prolog modules, which are referred to (unsurprisingly) as "Prolog modules".

### 2.5.1 Update 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.

A module contains one or more algorithms which it executes according to instructions from the controller. The algorithm language contains the basic imperative constructions, and allows calls to update rules and update rule classes as well as checks, queries and updates to the TIS.

TrindiKit provides language for writing module algorithms, called DME-ADL (Dialogue Move Engine Algorithm Definition Language[2]). It is also possible to implement modules directly in Prolog[3],

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.

### 2.5.2 Module interface variables

Typically, 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 Dialogue Move Engine

The term "Dialogue Move Engine" is used to designate the module or collection of modules which updates the information state based on observed dialogue moves, and for selecting moves to be performed. Abstractly, the DME can be seen as implementing a function

[2]While especially useful for writing DME module algorithms, this language can be used for implementing both DME and non-DME modules.

[3]Actually, modules can be implemented in any language which supports OAA, provided the TrindiKit is run using OAA.

from a collection of input dialogue moves and an ingoing information state to a collection of output dialogue moves and an outgoing state (see also Ljunglöf (2000). According to the definition of a DME, many parts of a dialogue system are not part of the DME. For example, modules which read or listen to input from the user, interpret utterances from the user in terms of dialogue moves, generate utterances from dialogue move specifications, or generate written or spoken output are not part of the DME.

The DME is not necessarily a separate agent, but often it is practical to implement it as one.

## 2.7 Controller

The controller wires the modules together using a control algorithm. It can also access the whole TIS. A TrindiKit system can be run either serially or asynchronously.

### 2.7.1 Serial control

In serial mode, the whole system is run as a single process and only one module at a time can be active. In this case, the control algorithm decides when to call a module algorithm, possibly based on the TIS. Only when the algorithm is finished executing can the controller decide what module to call next.

### 2.7.2 Asynchronous control

Asynchronous systems, as opposed to serial systems, allow different parts to run in parallel while still being interconnected. This is achieved by running (some of) the different parts as separate processes, or "agents". We will use "TrindiKit agent" to mean "a part of the TrindiKit running as an agent".

In asynchronous (concurrent) mode, the control algorithm specifies several *agents*[4] , each of which may contain one or more modules. Within each agent, the modules can be called according to a control algorithm similar to those used by the controller in serial mode. The control algorithm also includes *triggers* for each agent, which specifies under

[4]In the AE/OAA jargon (which we will adopt henceforth), an agent is a process interacting with other agents; a collection of interacting agents form an *agent community*. See also section 2.7.3.

what conditions an algorithm defined in one of the modules in the agent should run (as mentioned above, a module may specify more than one algorithm).

When run asynchronously, TrindiKit needs to run several components as agents. These include the TIS and the **print** agent, which prints rules and information state according to flag settings. Also, some components are used only when running asynchronously. These are:

- A **spy** agent which prints out some of the important events taking place: when a module algorithm is started and is complete, when and how the information state is updated etc.
- A **console** agent, where the user can inspect and interfere with the system components at runtime.

The behaviour of these agents can be customised by settings in the `concurrent.pl` file (see 3.7.3).

### 2.7.3 TrindiKit, AE, and OAA

When running asynchronously, TrindiKit utilises an agent communication protocol, which can be either the TrindiKit Agent Environment (AE) supplied with TrindiKit, or SRI's Open Agent Architecture (OAA). It is also possible to combine both environments in one system, by running TrindiKit on top of AE and having one or several modules simultaneously be OAA agents.

#### SRI's Open Agent Architecture

OAA (Open Agent Architecture) Martin *et al.* (1999), developed by SRI international, is "a framework for integrating a community of heterogeneous software agents in a distributed environment." [5] An agent is defined as

> ... any software process that meets the conventions of the OAA society. An agent satisfies this requirement by registering the services it can provide in an acceptable form, by being able to speak the Interagent Communication

[5] This quote is from the OAA web site, `http://www.ai.sri.com/~oaa/`.

Language (ICL), and by sharing functionality common to all OAA agents, such as the ability to install triggers, manage data in certain ways, etc. [6]

The services provided by an agent are expressed as a list of so called solvables (or "capabilities"). The solvable consists of three parts: the goal template, a list of permissions and a list of parameters. The goal template, which is the only part of solvable that is used in TrindiKit, describes (in ICL, which can be seen as a subset of Prolog) what kind of goal that can be solved. E.g., if an agent wants to share its ability to append two lists it declares a solvable with the goal template `append(_,_,_)`. In OAA, solvables may be registered, redeclared and removed at any time, even agents may register and disconnect from the OAA community at any time. However in TrindiKit, solvables are declared only when the system is started and are never withdrawn; all TrindiKit agents are started only once and disconnected only once (TrindiKit agents may not "enter the game" during execution).

Agents are coordinated by the so called facilitator. When an agent issues an `oaa_Solve` command, the facilitator will try to find an agent which can solve the goal and will then redirect the query to that agent. Additionally, requesting agents may choose to explicitly specify to which agent the goal should be sent. In TrindiKit, this kind of explicit addressing is always used (execution of goals are never routed through the facilitator). The primary reason is that this is usually more efficient. Explicit addressing is possible since TrindiKit is designed in such a way that the requesting agent always knows which agent to consult in a given situation.

### TrindiKit Agent Environment

AE (Agent Environment) can be seen as a stripped-down version of OAA. It was developed primarily for the TrindiKit.

AE has no facilitator. Instead, a so called AE starter is responsible for launching and coordinating a community of agents. A specific starter has to be built and run separately for each set of agents (as opposed to OAA, where the facilitator can coordinate any set of agents). In OAA it's possible to route goals through the facilitator, whereas in AE explicit addressing is compulsory.

In OAA agents can be distributed over a network of machines, whereas in AE all agents have to run on the same machine.

AE agents declare a set of services (using Prolog module-like notation, e.g. `append/3`).

[6] This quote is from the OAA website, `http://www.ai.sri.com/~oaa/`.

Services can be registered at any time but can not be withdrawn.

## 2.8 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.

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 agents which interact with the information state and are called upon by the controller. 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.

## 2.9 Flags

The flags are static while the system is running. They can be read by all modules and are set in the `start` file. There is a set of generic TrindiKit flags, but the user can add additional flags. All flags have a default value.

### 2.9.1 Display flags

- `format`; values: `text,latex,html`
  Output format for utterances, information states and rules.
- `show_state`; values: `all,is,no`
  Whether and how to display the information state.
  - `all`: display the whole TIS
  - `is`: display only the IS variable
  - `no`: do not display the information state
- `show_rules`; values: `yes,no`
  Whether to display update rules.

### 2.9.2 Debugging flags

These flags are used to turn debugging facilities on and off. These facilites are further described in section 3.10.

- `access_restrictions`; values: `on, off`
  Enforce restrictions on module access to TIS, as defined in modules (see 4.2.9). If an access restriction is violated, an error message will be displayed.
- `trace_rules`; values: `off,on`
  Trace the rules applied to the TIS.
- `console`; values: `yes,no`
  Run the console agent.
- `trace`; values: `yes,no`
  Run step-wise (conrurrent mode only).
- `spy`; values: `yes,no` Activate spy agent.
- `typecheck`; values: `yes,no`
  If set to `yes`, enforce typechecking[7]

[7]Not fully implemented in 2.0.

### 2.9.3 Serial/concurrent flag

This flags is set by the system according to the control algorithm. It should not be changed by the user.

- `architecture`; values: `serial,concurrent`

# Chapter 3

# TrindiKit functionality

Apart from the general architecture defined in 1.1 above, the TrindiKit provides

- definitions of datatypes, for use in TIS variable definitions
- a format for defining new datatypes
- simple default modules for input, interpretation, generation and output
- methods (checks, queries and updates) for accessing the TIS
- a language for specifying TIS update rules
- an update algorithm language for modules
- a control algorithm language for the controller, including concurrent control
- use of external resources
- a generic web demo generator
- methods for interfacing TrindiKit modules with OAA
- debugging facilities, including methods for visually inspecting the TIS

In this section we will give an overview of the structure of the TrindiKit implementation and any system implemented within the TrindiKit architecture.

## 3.1 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.

### 3.1.1 Datatypes

Datatypes provide a natural way of formalising information states. The TIS is specified using abstract data types, each permitting a specific set of queries to inspect the type and operations to change it.

Datatypes[1] are used extensively in the TRINDI DME architecture, most importantly for modeling the TIS.

A datatype definition includes the following:

1. Name
2. Operations, which modify objects
3. Conditions, which are true or false of objects

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.[2]

The following datatypes definitions are included in TrindiKit 2.0[3] (for definitions see Appendix A):

- ASSOCSET
- ATOM
- BOOL

[1] An alternative term is "datastructure".
[2] Typechecking is not yet fully implemented.
[3] Some definitions may be incomplete in the current release.

- COUNTER
- INTEGER
- MOVE
- PARTICIPANT
- PROGRAM_STATE
- QUEUE
- REAL
- RECORD
- SET
- STACK
- STACKSET
- STRING

To make use of a type definition, a *type declaration* is needed which declares a variable to be of a certain type.

### 3.1.2 General conditions and operations

Apart from the type-specific conditions and operations, there are some conditions and operations which apply to all 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*
- **val**(*Obj*) : the object unifies with *Obj* (identical to **unify**)
- **is_set** : the object is set (it is not **nil**)
- **is_unset** : the object is not set (it is **nil**)
- **empty_object**(*PseudoType*, *Var*) : *Var* is instantiated with an empty object of type *PseudoType*[4]

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 $T$

## 3.2 Defining new datatypes

The library of datatypes in TrindiKit 2.0 is still limited, and when building a system it may be necessary to define new datatypes, which may then be included in future releases of TrindiKit and reused in other systems. This section explains how this is done.

### 3.2.1 Datatype definition syntax

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**.

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 4.7). The TrindiKit will load the definitions into the `datatypes` Prolog module.

The "functor" or "operator" of a datatype condition or operation may be overloaded, i.e. several datatypes may have (syntactically) identical functors. For example, the functor `add` is defined for both sets and stacksets.

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 (1) 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`

Conditions and operations are optional, i.e. it is possible to define datatypes which have no conditions or operations.

### 3.2.2 Defining conditions

Conditions are defined as applying to an object of a certain type. Conditions may also have arguments. When calling the conditions from a rule or algorithm, the object can be indicated either by a TIS variable (which is evaluated to an object) or an explicit object. Conditions may also have arguments.

The format for condition definitions is shown in (1).

(1) `condition(`$Type, Cond$`(`$Arg_1, \ldots, Arg_n$`), `$Obj$`):-`$Body$

$Obj$ is an object of type $Type$. $Cond$ is the condition itself, e.g. `fst`, and $Arg_x$ is an argument. The optional $Body$ is any Prolog code needed to define the condition.

### 3.2.3 Defining operations

As with conditions, operations apply to an object of a certain type and they may have arguments. The difference is that an operation also specifies how the object changes as a result of successfully applying the operation. This is done by specifying an *output object*. This means that operations are defined relationally, i.e. as a relation between an input object, an operation (possibly taking arguments) and an output object.

While the operation definition is relational, and thus non-destructive, the successful *application* of an operation *is* destructive, i.e. it changes the information state. The destructivity consists in taking the input object and the operation, consulting the operation definition, receiving an output object, and replacing the input object with the output object.

The format for operation definitions is shown in (2).

(2) `operation(`$Type$`, `$Oper$`(`$Arg_1, \ldots, Arg_n$`),` $ObjIn$`,`
$ObjOut$ `):-` $Body$

$ObjIn$ and $ObjOut$ are objects of type $Type$. $Oper$ is the condition itself, e.g. `push`, and $Arg_x$ is an argument. The optional $Body$ is any Prolog code needed to define the operation.

### 3.2.4 A sample datatype definition

As an example, the TrindiKit definition of a stack is shown in (3). Note that the predicates used to define a datatype must be declared to be `multifile`.

(3)
```
:- 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) ).

% extend_top(ArgStack): extend ObjStack by attaching ArgStack to the top
% of ObjStack
% ObjStack := ObjStack0+ArgStack
operation( stack(_), extend_top(ArgStack), StackIn, StackOut ):-
        ArgStack = stack(ArgList),
        StackIn = stack(ListIn),
        append( ArgList, ListIn, ListOut ),
        StackOut = stack(ListOut).
```

### 3.2.5 Condition and operation macros

In addition to the conditions and operations provided by the datatype definitions, it is also possible to write *macros*. Macros define sequences of conditions (for *condition macros*) or operations (for *operation macros*). Like conditions and operations (but unlike rules), macros can take arguments.

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`.

- `macro_cond(` *Check*`,` *CheckList* `)`
  *CheckList* is a list of check calls; *Check* is true if all the checks in *CheckList* are

true.

- `macro_op(` *Update*`,` *UpdateList* `)`
  *UpdateList* is a list of updates and/or queries; to apply *Update*, apply all the updates and queries in *UpdateList*.

A sample condition macro is shown in (4). It defines the condition macro `not_all_integrated(Path)` which is true if there is some element in the association set at `Path` which is associated with `false`.

(4) (in macro file:)

```
macro_cond( not_all_integrated(Path),
            [ in#rec(Path, M) and
            ( not assoc#rec(Path,M,true ) )
            ] ).
```

A sample operation macro is shown in (5). It defines an operation `pop_to(Path1,Path2)` which pops the stack at `Path1` and pushes the popped element to the stack at `Path2`.

(5) (in macro file:)

```
macro_op( pop_to(Path1, Path2),
            [ ! fst#rec(Path1, X),
              pop#rec(Path1),
              push#rec(Path2,X)
            ] ).
```

### 3.2.6 Datatype printers

Optionally, one may want to give formatting instructions for a datatype. These are used for printing objects of that type in a given format. TrindiKit 2.0 includes facilities for generating text, HTML, and LaTeX output[5]. For any file *Dataype*`.pl` defining a datatype, TrindiKit will search for a file `print_textitdatatype.pl` containing formatting instructions for that datatype.

If no printer is defined for a datatype in a certain format, the internal representation, e.g. `stack([a,b,c])` will be printed.

[5]Additional formats may be included in later releases.

## 3.3 Methods for accessing the TIS

The TIS can be accessed in three ways: *checks*, *queries*, and *updates*. Checking and querying are ways to find out what the information state is like, and they can bind Prolog variables. Updates change the information state, but can not bind Prolog variables. Check-calls are true or false, whereas query and apply-calls fail or succeed.

When using DME-ADL or Control-ADL, TrindiKit will handle the TIS access. To build a module which does not use ADL one needs to access the TIS using predicates exported from the TIS handler. In Prolog, the TIS is read using the predicates `check_conditions/1`, whose argument is a list of checks on the TIS, or `check_condition/1` whose argument is a single check. It is written to using `apply_operations/1`, whose argument is a list of queries and/or updates on the TIS, or `apply_operation/1`.

### 3.3.1 Basic check syntax

Checks can use either condition or operation definitions. Checks on operations use the `result` construction.

#### Checks using conditions

Given a condition $Cond(Arg_1, \ldots, Arg_n)$ defined for objects of type $Type$, checks have the following basic formats:

- **(C1)** $Obj$ :: $Cond(Arg_1, \ldots, Arg_n)$
- **(C2)** $Cond(Obj, Arg_1, \ldots, Arg_n)$

At call time, i.e. when the check is being made, $Obj$ can be either an object (a Prolog term) or a TIS variable of type $Type$ . In the latter case, TrindiKit will replace the TIS variable with the object which is the value of the variable before consulting the condition definition[6].

[6]There is no explicit evaluation marker in TrindiKit 2.0, so any term which is identical to a TIS variable will be evaluated as a TIS variable.

#### Checks using operations

It is possible to check the result of an operation. Given an operation $Oper(Arg_1, \ldots, Arg_n)$ defined for objects of type $Type$, checks can the following basic formats:

- **(C3)** `result(`$InObj$`,` $OutObj$`) ::` $Oper(Arg_1, \ldots, Arg_n)$

This syntax makes it possible to apply operations to objects which are not in the TIS but are stored in a Prolog variable. Check-calls which use operations can backtrack if the operation is nondeterministic[7].

#### IS shortcut syntax

For checks on the IS there is a special shortcut syntax available.

- **(C4)** $Cond(Arg_1, \ldots, Arg_n)$

This is equivalent to `is ::` $Cond(Arg_1, \ldots, Arg_n)$.

#### Using condition macros

Condition macros can be used in check calls. The syntax is the same as in the definition of the macro, i.e. given a condition macro definition `cond_macro(`$CondMacro(Arg_1, \ldots, Arg_N)$`)`, the syntax for a check call to this condition is this:

- **(CM)** $CondMacro(Obj, Arg_1, \ldots, Arg_n)$

### 3.3.2 Checks and First Order Logic

Given check calls $Check$, $Check_1$ and $Check_2$, the following constructs are available as check calls:

[7] As noted in 3.2.3, operations in TrindiKit are not "real" operations since they can be nondeterministic, i.e. have several possible results

- $Check_1$ `and` $Check_2$
  This is equivalent to $Check_1$ `,` $Check_2$; `and` is right associative.

- $Check_1$ `or` $Check_2$
  This is true if $Check_1$ is true or $Check_2$ is true. $Check_1$ will be tested first, and only if it is false will $Check_2$ be tested.

- `not` $Check$
  This is true if $Check$ is false.

- `forall(`$Check_1$`,` $Check_2$`)`
  This is equivalent to `not (`$Cond_1$ `and (not` $Cond_2$`))`.

In terms of First Order Logic (FOL), a precondition list can be seen as a conjunction of propositions (which are true or false of the TIS), existensially quantified over all Prolog variables occuring in the checks - except for variables occuring only within the scope of negation or universal quantification.

An illustration is shown in (6) ($C_1(X)$, $C_2(X,Y)$ and $C_3(Y,Z)$ are check calls; $X$, $Y$ and $Z$ are Prolog variables, and $x, y$ and $z$ are the corresponding FOL variables).

(6) `(` $C_1(X)$ `and` $C_2(X,Y)$ `and` $C_3(Y,Z)$`)` $\sim$
$\exists x, y, z(C_1(x) \wedge C_2(x,y) \wedge C_3(y,z))$

**Negation**

When evaluating `not` $Check$, some Prolog variables appearing in $Check$ may already have become bound (when evaluating a previous check). Any previously bound variables appearing in $Check$ will still be bound when evaluating `not` $Check$.

(7) `(`$C_1(X)$`and (not` $C_2(X)$`)) and` $C_3(Y,Z)$`)` $\sim$
$\exists x, y, z(C_1(x)) \wedge \neg C_2(x) \wedge C_3(y,z)$

Any previously *unbound* variables appearing in $Check$ will not be bound by checking `not` $Check$. They are interpreted as existensially quantified within the scope of the negation, as illustrated in (8). Any occurrences of these variables occuring in following conditions will be independent, i.e. they can be regarded as separate variables.

(8) `(`$C_1(X)$ `and (not` $C_2(X,Y)$`)) and` $C_3(Y,Z)$`]` $\sim$
$\exists x, y, z(C_1(x)) \wedge \neg\exists y'(C_2(x,y')) \wedge C_3(y,z)$

**Universal quantification**

The binding behaviour of Prolog variables inside the scope of `forall` is similar to that for `not`, which is natural since `forall` is defined using `not`.

If $X_1, \ldots, X_n$ are variables in $Check_1$ which are not previously bound, and $Y_1, \ldots, Y_m$ are variables in $Check_2$ which are not previously bound *and* do not occur in $Check_1$, the FOL interpretation is as shown in (9).

(9) `forall(`$Check_1, Check_n$`)` $\sim$
$\forall x_1, \ldots, x_n(Cond_1 \rightarrow \exists y_1, \ldots, y_m(Check_2))$

Any previously bound variables appearing in $Check_1$ or $Check_2$ will still be bound when evaluating `forall(`$Check_1, Check_2$`)`. Any previously *unbound* variables appearing in $Check_1$ or $Check_2$ will not be bound by checking `forall(`$Check_1, Check_2$`)`. Any occurrences of these variables occuring in following conditions will be independent, i.e. they can be regarded as separate variables.

### 3.3.3 Shorthand syntax for identity and unification checks

The general conditions `unify` and `equal` can be used to check if any two objects unify or are identical, respectively. The basic syntax for `unify` and `equal` checks are

- $Obj$`::unify(`$Arg$`)`
- $Obj$`::equal(`$Arg$`)`

For cases where $Obj$ is an explicit object $Arg_0$ (rather than a TIS variable) the following shorthand syntaxes are available:

- $Arg_0$`=`$Arg$, equivalent to $Arg_0$`::unify(`$Arg$`)`
- $Arg_0$`\=`$Arg$, equivalent to `(not` $Arg_0$`::unify(`$Arg$`))`
- $Arg_0$`==`$Arg$, equivalent to $Arg_0$`::equal(`$Arg$`)`
- $Arg_0$`\=`$Arg$, equivalent to `(not` $Arg_0$`::equal(`$Arg$`))`

For cases where $Obj$ is a TIS variable $Var$ the following shorthand syntaxes are available:

- $Var$`$=`$Arg$, equivalent to $Var$`::unify(`$Arg$`)`
- $Var$`$==`$Arg$, equivalent to $Var$`::equal(`$Arg$`)`

### 3.3.4 Query syntax

Syntactically, queries are checks prepended by "!". This operator transforms a condition into a query. A query `!` $Check$, where $Check$ is a check call, will fail if Check is not true. When a query fails, an error is reported. There is no backtracking over queries, i.e. if a query fails it will not be tried again. This is the only difference between checks and queries.

This means that given a check call $Check$, a query call has the form

- **(Q)** `!`$Check$

### 3.3.5 Basic update syntax

Updates use operation definitions which have the general form $Oper(Arg_1, \ldots, Arg_n)$. Each update call modifies an object which is the value of a variable $Var$ in the TIS, by evaulating $Var$, passing it as the input object to the operation definition, together with $Arg_1, \ldots, Arg_n$, and finally setting the value of $Var$ to be the output object.

Updates have the following basic formats:

- **(O1)** $Var$ `::` $Oper(Arg_1, \ldots, Arg_n)$
- **(O2)** $Oper(Obj, Arg_1, \ldots, Arg_n)$

**IS shortcut** For updates to the IS there is a special shortcut syntax available.

- **(O3)** $Oper(Arg_1, \ldots, Arg_n)$

This is equivalent to `is ::` $Oper(Arg_1, \ldots, Arg_n)$.

### Using operation macros

Operation macros can be used in check calls. The syntax is the same as in the definition of the macro, i.e. given a operation macro definition `op_macro(`$OpMacro(Arg_1, \ldots, Arg_N)$`)`, the syntax for an apply-call to this operation is this:

- **(OM)** $OpMacro(Obj, Arg_1, \ldots, Arg_n)$

### The `forall_do` construction

It is possible to apply an operation repeatedly using a single update call. This is done using the syntax in (10), where *Check* is a check call and *Update* is an update call.

(10) `forall_do(`*Check*`,` *Update*`)`

Let $X_1, \ldots, X_n$ be all unbound Prolog variables of *Check* (i.e. those which are not bound when the update is applied). Now, the interpretation of (10) goes as follows: "For all bindings of $X_1, \ldots, X_n$ which make *Check* true, apply *Update*."

As an example, for all $A$ such that $A$ is in the set `latest_moves`, (11) will add a pair `(`$A$`,false)` to the set at `shared^lu^moves`.

(11) `forall(in(latest_moves,A),add#rec(shared^lu^moves,(A,false)))`

NOTE: Any unbound Prolog variables appearing in *Check but not in Update* must be prefixed with a "_". Otherwise, *Update* will only be applied for the first solution of *Check*.

### Prolog variables in the TIS

TrindiKit does not prohibit Prolog variables as part of the TIS, i.e. objects can contain Prolog variables. This has the advantage that it is possible to have partially instantiated objects (*non-ground terms* in Prolog terminology) which may become fully instantiated at a later point.

However, this also makes it possible for checks and queries to temporarily change the information state by unifying a partially instantiated objects in the TIS with a more specific

object. For example, if the value of the TIS variable `latest_move` is `ask(happy(X))`, checking `latest_move $= ask(happy(john))`will have the effect that `latest_move` now has the value `ask(happy(john))`.

If this check is part of the preconditions list of a rule, and a later precondition fails, TrindiKit may backtrack which may result in `X` becoming unbound again. These bindings "survive" within the scope of an update rule or a sequence of checks called using using `check_conditions`. After the rule or sequence of checks is done, any Prolog variables in the TIS will again be unbound.

## 3.4 Rule definition format

Update rules are rules for updating the TIS. They consist of a rule name, a precondition list, and an effect list. Preconditions are check-calls, and effects can be queries or update-calls.

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.

The rule definition format is shown in (12).

(12) `rule(` *RuleName*`,` *PrecondList*`,` *EffectsList* `)`
`of_class(` *RuleName*`,` *RuleClass* `)`

Here, *PrecondList* is a list of check-calls[8] (Section 3.3.1) and *EffectsList* is a list of queries and apply-calls (Section 3.3.5). The *RuleClass* may be used in defining DME algorithms (Section 3.5.

A sample update rule definition is shown in (13).

[8] TrindiKit 2.0 actually allows queries in rule preconditions. However, this is not encouraged since it invalidates the interpretation of preconditions in terms of FOL. Future TrindiKit versions may not support this.

(13)
```
rule( integrateUserQuestion,
      [ 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 )
```

### 3.4.1 Backtracking and variable binding in rules

When a rule is applied to the TIS, the preconditions will be evaluated in the order they appear. Since conditions and operations may be nondeterministic, checks containing Prolog variables may have several possible results. For example, checking `member(X)` on a set `set([a,b,c])` has `X=a`, `X=b` and `X=c` as possible results.

The first time this check is made, the first solution will be returned, i.e. `X` will become bound to `a`. If a later precondition which uses `X` is not true for `X=a`, TrindiKit will use Prolog's backtracking facility to go back and get the second solution `X=b`. In this way, the TrindiKit rule interpreter will try to find a way to bind the Prolog variables appearing in the precondition list[9] so that all the preconditions hold.

Once this has succeeded, the effects of the rule will be applied using the variable bindings obtained when checking the preconditions. Continuing the example above, if all preconditions succeed with `X` bound to `a`, this binding will "survive" to the effects and any appearance of `X` in the effects will be equivalent to an appearance of `a`.

### 3.4.2 Rules are protected from asynchronous intervention

Rules can be seen as small algorithm parcels sent to the TIS handler, where they are executed (applied). Since the TIS can only execute one rule at a time, rules are protected from asynchronous interventions from other modules. This means that if the preconditions of a rule are true when they are checked, they will still be true when the effects are applied.

[9] apart from those appearing in the scope of a negation or universal quantification, see 3.3.2 and 3.3.2 respectively

### 3.4.3 Preconditions and First Order Logic

The precondition list `[`$Check_1, \ldots, Check_2$`]` in a rule is equivalent to a conjunction `(`$Check_1$ `and` ... `and` $Check_n$`)`. In terms of First Order Logic (FOL), a precondition list can be seen as a conjunction of check-calls (which are true or false), existensially quantified over all Prolog variables occuring in the checks - except for variables occuring only within the scope of negation or universal quantification. See also 3.3.2.

## 3.5 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 check call; $R$, $S$ and $T$ are algorithms, *Rule* is the name of an update rule, and *RuleClass* is a rule class):

1. *Rule*
   apply the update rule *Rule*

2. *RuleClass*
   apply an update rule of class *RuleClass*; rules are tried in the order they are declared

3. `[`$R_1, \ldots, R_n$`]`
   execute $R_1, \ldots, R_n$ in sequence

4. `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`

5. `while` $C$ `do` $R$
   while $C$ is true of the TIS, execute $R$ repeatedly

6. `repeat` $R$ `until` $C$
   execute $R$ repeatedly until $C$ is true of the TIS

7. `repeat` $R$
   execute $R$ repeatedly until it fails; report no error when it fails
   **Example:** `repeat refill`

8. `repeat+` $R$
   execute $R$ repeatedly, but at least once, until it fails; report no error when it fails

9. `try` *R*
   try to execute *R*; if it fails, report no error

10. *R* `or` *S*
    Try to execute *R*; if it fails, report no error and execute *S* instead

11. `test` *C*
    if *C* is true of the TIS, do nothing; otherwise, halt execution of the current algorithm

12. *SubAlg*
    execute subalgorithm *SubAlg*

Subalgorithms are declared using `==>`, which is preceded by the subalgorithm name and followed by the algorithm, as in (14).

(14)
```
main_update ==> [ grounding, repeat+ ( integrate or
accommodate ) ].
```

A sample DME-ADL algorithm is shown in (15).

(15)
```
if (latest_moves $== failed)
  then (repeat refill)
  else (     [ grounding,
               repeat+ ( integrate or accommodate  ),
               (   if (latest_speaker $== usr)
                      then ([ (repeat refill),
                              (try database) ])
                      else store
               )
             ] ).
```

### 3.5.1 Backtracking behaviour for checks, queries and updates in module algorithms

In concurrent mode, checks calls in update algorithms *do not backtrack*[10].

In serial mode, conjunctions of check calls display the same backtracking behaviour as conjunctions or sequences of check calls in update rules (see 3.4).

[10] In a future realease of TrindiKit we hope to make it possible for check calls to backtrack in concurrent mode.

Update calls and rule calls can fail, which will result in an error message. If a rule class is called, the intpreter will try to find a rule of that class which succeeds; if no such rule is found, the call fails.

### 3.5.2 Rule calls in module algorithms

Module algorithms are executed locally by the ADL interpreter. Checks, queries, updates and rule calls in the module algorithm are passed to the TIS where they are executed. Rules can be seen as small algorithm parcels sent to the TIS. Since the TIS can only execute one rule at a time, rules are protected from asynchronous interventions from other modules.

## 3.6 Provided modules

The TrindiKit package includes a couple of simple modules which can be used to quickly build prototype systems. Currently, four modules are included:

- `input_simpletext`: a simple module which reads text input from the user and stores it in the TIS
- `output_simpletext`: a simple text output module
- `intpret_simple1`: an interpretation module which uses a lexicon of key words and phrases to interpret user utterances in terms of dialogue moves
- `generate_simple1`: a generation module which uses a lexicon of mainly canned sentences to generate system utterances from moves

The provided modules are further described in appendix B.

We hope to include modules for speech input and output in future TrindiKit releases. We also encourage building modules which may be re-used in other systems and included in future TrindiKit releases.

## 3.7 The Control-ADL language

The control algorithm specifies whether a system should be run in serial or asynchronously. Serial algorithms are simpler than asynchronous ones, and the syntax used for asynchronous algorithms subsumes the syntax for serial algorithms.

### 3.7.1 Serial control algorithm syntax

The serial Control-ADL language is identical to the DME-ADL language, except that it calls module algorithms instead of rules, and it can include the `print_state` instruction. Each algorithm has a name, and each module may define one or more algorithms.

The available modules are specified by the `selected_modules` predicate in the system configuration file (see Section 4.7).

Serial control algorithms are expressions of any of the following kinds ($C$ is a TIS condition; $R$, $S$ and $T$ are serial control algorithms, $Module$ is a module, and $Alg$ is an update algorithm):

1. $Module$:$Alg$
   Call algorithm $Alg$ in module $Module$; This requires that $Module$ exports and defines $Alg$`/0`

2. $Module$
   Equivalent to $Module$:$Module$

3. `[`$R_1, \ldots, R_n$`]`
   execute $R_1, \ldots, R_n$ in sequence

4. `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`

5. `while` $C$ `do` $R$
   while $C$ is true of the TIS, execute $R$ repeatedly

6. `repeat` $R$ `until` $C$
   execute $R$ repeatedly until $C$ is true of the TIS

7. `repeat` $R$
   execute $R$ repeatedly until it fails; report no error when it fails
   **Example:** `repeat refill`

8. `repeat+` $R$
   execute $R$ repeatedly, but at least once, until it fails; report no error when it fails

9. `try` $R$
   try to execute $R$; if it fails, report no error

10. $R$ `or` $S$
    Try to execute $R$; if it fails, report no error and execute $S$ instead

11. `test` $C$
    if $C$ is true of the TIS, do nothing; otherwise, halt execution of the current algorithm

12. *SubAlg*
    execute subalgorithm *SubAlg*

13. `print_state`
    Print the TIS or IS, provided the value of the `show_state` is `all` or `is`, respectively

Subalgorithms are declared using `==>`, which is preceded by the subalgorithm name and followed by the algorithm.

A sample serial Control-ADL algorithm is shown in (16).

(16)
```
[reset,
 repeat ( [ select,
            generate ,
            output,
            update,
            print_state,
            test( program_state $== run ),
            input,
            interpret,
            print_state,
            update,
            print_state
          ] )
]
```

### 3.7.2 Concurrent control algorithm syntax

A concurrent (asynchronous) control algorithm in Control-ADL specifies a number of *update agents*. Each update agent uses one or several modules, each specifying one or more

update algorithms. The update agent may either call these algorithms directly, or it may specify a serial control algorithm (actually, the former is a special case of the latter where the serial control algorithm is just a call to a module algorithm). The concurrent control algorithm must also specify one or several triggers, specifying the conditions under which to run an update algorithm in a module or a serial control algorithm.

It may also include module importation calls. By default, a module is not imported into an agent until an algorithm in that module is triggered. However, it is possible to import a module into an agent at startup, using `import` $Module$.

A concurrent control algorithm has the following general format ($Agent_i$ is an agent name):

(17) `control_algorithm((` $Agent_1$ | $Agent_2$ | ... | $Agent_n$ `))`
Run Agent 1 to $n$ asynchronously

An agent declaration has the following form, where $SerialAlgorithm_j$ are serial control algorithms:

(18) $AgentName$: { `import` $Module_1$`,`
...
`import` $Module_p$`,`
$Trigger_1$ `=>` $SerialAlgorithm_1$`,`
...
$Trigger_m$ `=>` $SerialAlgorithm_m$ }.
When $Trigger_j$ is true, run $SerialAlgorithm_j$.
$(1 \leq j \leq m)$.

$AgentName$ is a unique label for that agent - two agents may not have the same name. There must be at least one trigger and one algorithm call (i.e. $1 \leq m$) but there may be zero or more module importations (i.e. $0 \leq p$). The serial control algorithms are defined using Control-ADL, as described in 3.7.1. As mentioned in 3.7.1, calls to algorithms can be made in two different ways: $Module$:$Algorithm$ or just $Module$, which is interpreted as $Module$:$Module$.

Warning: Be careful when using `repeat` statements in update algorithms or serial control algorithms called from an asynchronous control algorithm, since they will never leave the loop and will therefore never trigger more than once.

A trigger $Trigger_j$ can have any of the following formats:

- `condition(` $C$ `)`

$C$ is a *trigger condition*, which has any of the following formats (where $Var$ is a TIS variable):

- `val(` $Var$`,` $Val$ `)`
  $Var$ becomes set to $Val$.
- `is_set(` $Var$ `)`
  Var changes value.
- `is_unset(` $Var$ `)`
  $Var$ becomes unset (i.e. $Var$ becomes set to `nil`).
- `empty(` $Var$ `)`
  $Var$ becomes empty (i.e. $Var$ becomes set to `nil`.
- `not val(` $Var$`,` $Val$ `)`
  The value of $Var$ becomes set to some value other than $Val$.
- `not empty(` $Var$ `)`
  $Var$ becomes set to some value other than `nil`.

- `init`
  This trigger is activated at start-up.
- `new_data(` *Stream* `)`
  The Prolog stream *Stream* (e.g. `user_input`) has new readable data.

A sample concurrent control algorithm is shown in (19).

(19)
```
control_algorithm((

input: {
   init => input:display_prompt,
   new_data(user_input) => input
} |

interpretation: {
  import interpret,

  condition(is_set(input)) =>
     [ interpret, print_state ]
} |

dme: {
  import update,
  import select,

  init => [ select ],
  condition(not empty(latest_moves)) =>
   [
     update,
     if val(latest_speaker,usr) then
       select
     else
       []
   ]
} |

generation: {
  condition(is_set(next_moves)) =>
     generate
} |

output: {
  condition(is_set(output)) =>
     output
}

)).
```

### 3.7.3 Settings for concurrent systems

When running a concurrent TrindiKit system, certain settings must be specified in the file `concurrent.pl`. These are:

- Choice of agent communication protocal. TrindiKit can use either AE (Agent

Environment) or SRI's OAA (open Agent Architecture) as agent communication protocol.

- Appearance settings for agent windows
- Settings for the `spy` agent, determining what interactions to print
- Resource modes for each resource, i.e. `tis`, `module` or `standalone` (see 3.8.3)

## 3.8 Resources

It is often necessary to use external resources, such as databases, domain knowledge, lexica etc. in a dialogue system. TrindiKit's reosurce interface definitions makes it possible to hook up resources to the TIS and access them in the same way as any other object in the TIS.

### 3.8.1 Resource objects and Resource Interface Variables

In TrindiKit, resources are viewed as objects of certain type. A resource interface definition specifies the type of the interface object by defining conditions and operations on objects of that type. Resources can be called directly, if the resource call explicitly states a resource object, or via a resource intervace variable (RIV) which is part of the information state. The latter method makes it possible to switch resources by changing the value of the RIV.

While this may require some additional work, it has the advantage that the TIS as a whole can be regraded as a typed object. Also, it gives the user the possibility to define resources as static (if the resource interface definition only specifies conditions) or dynamic (if it also specifies operations).

It is very important to make sure that resources do not try to access the TIS, including other resources. Otherwise, deadlocks may result. Also, running modules in `module` or `standalone` will not work.

### 3.8.2 Resource definition syntax

There may be several resources of each type; for example, there may be lexica for several languages which all are of the type `lexiconT`. To declare that some resource object has a

certain type, `of_type/2` is used.

As an example, the interface definition for the GoDiS lexicon is shown in (20).

(20) (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( lexiconT ).

of_type( lexicon_travel_english, lexiconT ).
of_type( lexicon_autoroute_english, lexiconT ).
of_type( lexicon_travel_swedish, lexiconT ).

resource_variable_of_type( lexicon, lexiconT ).

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

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

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

Here, the `lexicon` TIS variable has type `lexiconT` and there are two objects of the `lexiconT` type: `lexicon_travel_swedish` and `lexicon_travel_english`, corresponding to the Swedish and English travel domain lexicons, respectively. To select a lexicon, the `lexicon` variable in the TIS is set to the appropriate value. If no resource variable is defined, the resources have to be named explicitly in the resource calls.

The resource filename should be the same as the resource name. To use a resource in a system, the resource name must be included in the `selected_resources` declaration in the system specification file.

### 3.8.3 Running resources in `module` and `standalone` mode

If a resource variable is used in a resource call, the TIS handler evaluates the interface variable in question, and passes the condition/operation on the selected resource object

which the value of the resource interface variable. This enables dynamic switching between resources during dialogue, e.g. to move to a new domain or change language.

When a system is run asynchronously, resources can be accessed via the TIS handler in the same way as when running serially; the only difference is that each agent will have its own TIS handler which communicates with the central TIS. In some cases, there may be a lot of data traffic to the central TIS, resulting in a slow system. To help avoid this, TrindiKit makes it possible to hook up resources directly to modules, or to run the resource as a standalone process.

How a resource is handled is specified in the user-supplied file `concurrency.pl`. Here, any resource can be defined to run in mode `tis`, `module` or `standalone` using `resource_mode/2`:

- `resource_mode(`*Resource*`, tis)`: *Resource* is accessed via the central TIS.
- `resource_mode(`*Resource*`, module)`: *Resource* is accessed directly from the module. In this mode, the resource will automatically be imported into all modules which use it.
- `resource_mode(`*Resource*`, standalone)`: *Resource* is run as a separate process, and can thus be called from several modules.

If a resource is static, there is no reason to run it in standalone mode; instead, it should be run in module mode if TIS data traffic is a problem. Since the resource cannot be changed, there is no harm in letting each module make its own copy of the resource. However, if a dynamic module is to be used by several modules running in different processes, it should not be run in module mode, as this might result in there being several copies of a resource which are not mutually consistent.

Since all update rules are executed by TIS, resource calls in rules will always be routed to the central TIS. This means that there is no efficiency gain for resource calls in update rules; only resouce calls made in module algorithms benefit from declaring a module as being of type `module` or `standalone`.

Warning 1: resource calls which are made directly from a module algorithm (i.e. not from a rule) should not use resource interface variables, since these must always be evaluated by the central TIS. When calling a resource running in `module` or `standalone` mode from a module, the resource object should therefore be stated explicitly. If one wants to be able to switch between such resources by changing the value of an RIV, some special tricks must be employed. For example, the module could have an small piece of Prolog code which is triggered every time the relevant RIV is changed, and asserts the new value of the RIV internally. Before every call to the resource, this internal asserted clause would have to be

consulted to find the current value of the RIV. This amounts to the module "subscribing" to an RIV. We plan to include support for this in future verisions of TrindiKit.

Warning 2: Check calls to standalone resources will not backtrack. Instead, the same result will be returned repeatedly.

#### Resource names and resource variable values

In some cases, one may want to put together the value of the resource variable in runtime, e.g. depending on the value of flags. To make this easier, all occurances of "-" in the value of a resource variable will be replaced by "_". The latter symbol can appear in filenames and module names, whereas the former cannot; the former can be used to connect Prolog terms, whereas the latter cannot.

## 3.9 The generic Web-demo

The TrindiKit generic web-demo generator will generate CGI-scripts etc., enabling interaction with the system via a webpage. The Web-demo is described in Appendix D.

## 3.10 Debugging facilities

Since different debugging facilities are available in serial and asynchronous modes, these will be described in separate sections.

### 3.10.1 Serial mode

In serial mode, all interaction with the system takes place in a single window. The TIS and the operations applied to it will be printed according to the settings of the `print_rule` and `print_state` flags (see section 2.9). When the system is halted, it is possible to inspect and alter the TIS and flags.

#### Accessing the TIS

In serial mode, 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(+`$Var$`, -`$Val$`)`.

#### Accessing flags

Before the system is started, or if it has been halted, the predicate `flags/0` prints all flags, their current values and a short description of each flag. The value of a flag is changed by `setflag(`$Flag$`,` $Value$`)`.

### 3.10.2 Asynchronous mode

In asynchronous mode, interaction with the system takes place in several windows at once. Each agent may have its own window, but only some allow direct interaction with the user. There are three agents providing functions which are useful for debugging: the **console**, **print** and **print** agents.

#### The console agent

The following functionalities are available in the console agent:

- Send commands to agents using the form $Agent::Command$ where $Command$ can be

  - `do(` *Query* `)`: *Query* is sent to the agent for execution. This is a non-blocking call, i.e. the console agent does not expect any answer from the agent
  - `solve(` *Query* `)`: *Query* is sent to the agent, and the response is displayed in the **console** agent window.

- List agent information:
  - `info` - gives a list of all agents and their services
  - `info(`*Agent*`)` - show information about a specific agent, including the services it offers

### Inspecting the TIS using the print agent

In asynchronous mode, the TIS and operations applied to it aredisplayed in the `print` agent according to the settings of the `print_rule` and `print_state` flags (see section 2.9).

The print agent can also be controlled directly from the console agent using the following command:

- `print :: do(print_state)`: print the TIS according to the setting of the `show_state` flag.

### Manually checking and updating the TIS

TIS variables can be set and evaluated in the `console` agent window using the follwing constructions, respectively:

- `tis::solve(apply_operation(set(+`$Var$`, +`$Val$`))).`
- `tis::solve(check_condition(val(+`$Var$`, -`$Val$`))).`

### Accessing flags

In asynchronous mode, flags can only be accessed before the system is run. As in serial mode, the predicate `flags/0` prints all flags, their current values and a short description of each flag. The value of a flag is changed by `setflag(`$Flag$`, `$Value$`)`.

### The spy agent

If the `spy` flag is set to `yes`, the spy agent window will appear when running the system. The spy agent displays triggers and operations applied to the TIS, and indicates which module is responsible for each operation.

### Running step by step

In the asynchronous mode, it is possible to run a system one step at a time. To switch this option on, set the `trace` flag to `yes` before running the system.

# 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.

## 4.1 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 the information state object has a certain type is specified using `infostate_of_type/2`.

### 4.1.1 Specifying information state type

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

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

```
:- module(infostate, [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.

### 4.1.2 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.

(22) (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 ).
```

### 4.1.3 Specifying resource interface variables

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

### 4.1.4 Defining macros

Macros are implemented using the syntax described in 3.2.5. To specify which file contains the macros, use `selected_macro_file`/1, as in (23).

(23) (in system configuration file:)

`selected_macro_file(` *FileName* `).`

Given this declaration, TrindiKit will load the macros in *FileName*`.pl`.

## 4.2 Building modules

A 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 module consists of the following parts:

- a module name declaration
- importation of `tis_access`
- loading of rules
- (optionally) loading of the DME-ADL interpreter
- the module algorithm or algorithms
- the algorithm predicate or predicates
- access restrictions (in the case of non-DME modules)

The Prolog file which defines a TrindiKit module can have any name. The link between a TrindiKit module and the file where it is implemented is declared in the system specification file by the `selected_modules/1` predicate.

### 4.2.1 Module name declaration

The module is declared as a Prolog module which exports the predicates used to call the algorithms specified by the module (see Section 4.2.7). *The name given to the Prolog module will also be the name of the* TrindiKit *module which the file implements.* This is also the name used in the control algorithm to refer to the TrindiKit module, and in `selected_modules/1` in the system specification file.

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

### 4.2.2 Importing TIS access

Every TrindiKit module must load `tis_access` to be able to access to TIS.

(25) `:- use_module(library(tis_access)).`

### 4.2.3 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)).`

### 4.2.4 Load the DME-ADL interpreter

To use the DME-ADL interpreter, it must be loaded into the module using the clause in (27).

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

### 4.2.5 The update algorithm

For instructions how to write update algorithms, consult 3.5.

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
                           )
                         ] ) ).
```

### 4.2.6 Writing Prolog algorithms

Modules can either use the DME-ADL interpreter, or be written in plain Prolog. In the latter case, no special restrictions apply. Check calls to the TIS are made using the `check_condition/1` predicate. Queries and updates to the TIS are applied using `apply_operation/1`. Rules are applied using `apply_rule/1`.

### 4.2.7 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 ).
```

### 4.2.8 Writing update rules

If a module makes uses rule calls, it requires a file (Prolog module) where the update rules are specified. 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 explained in 3.4

### 4.2.9 Access restrictions for non-DME modules

DME modules (those listed in the system configuration file using dme_modules/1) have access to the whole TIS. Non-DME modules have limited access to the TIS, and are ofthen only 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 (31), this module can read and write only to the input variable.

(31)
```
write_access( [input] ).
read_access( [input] ).
```

## 4.3 Building a controller

The controller contains the control algorithm, written using Control-ADL[1]. It can inspect specified parts of the TIS. The control module access to the whole TIS. (see Section 4.2.9).

A controller specification consists of the following parts:

- importation of operators

[1]Control-ADL will be loaded automatically into the controller.

- the control algorithm itself

The controller should be defined in the file `control.pl`.

### 4.3.1 Importing operators

The predicates in (32) must be imported into the control module.

(32)
```
:- use_module(library(control_operators)).
:- use_module(library(tis_operators)).
```

### 4.3.2 The control algorithm

The Control-ADL language which is used to define the control algorithm is explained in 3.7.

## 4.4 Building resources

The file which defines the resource must include a module declaration, as in (33). The module declaration must list all predicates that are to be exported and used in TIS conditions and operations.

(33)
```
:- module( lexicon_travel_english, [output_form/2, input_form/2, yn_answer/1] ).
```

The Prolog file defining a resource should have the same name as the resource.

## 4.5 Building resource interfaces

Resources must be be connected to the TIS via a resource interface definition. The syntax for resource interface definitions is explained in 3.8.2).

Apart from specifying resource interface variables, the file `resource_interfaces.pl` also exports the predicates `resource_condition/3`, `resource_operation/4` and `is_resource_type/2` to the `datatypes` component.

The resource (a Prolog module) must export the predicates needed for giving a type declaration of the corresponding resource interface variable.

## 4.6 Adding 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 (34).

(34) (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' ).
```

## 4.7 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.

### 4.7.1 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.

(35) (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.

(36) (in system configuration file:)

```
selected_macro_file( godis_macros ).
```

### 4.7.2 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.

(37) (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
                 ]).
```

### 4.7.3 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.

(38) (in system configuration file:)

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

### 4.7.4 Loading and selecting resources

The predicate `selected_resources/1` is specifies a list of resources that should available for hookup to the TIS. Each item *ResourceFile* in the list corresponds to a a file *ResourceFile*`.pl` in the search path. The file defines a resource object with the same name as the file, implemented as a Prolog module.

(39) (in system configuration file:)

```
selected_resources( [database_travel,
                     database_autoroute,
                     lexicon_travel_english,
                     lexicon_travel_svenska,
                     lexicon_autoroute_english,
                     domain_travel,
                     domain_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 $F$.

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 (40) will select these resources by setting the resource variables to the appropriate values.

(40) (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.

### 4.7.5 Specifying initial TIS

When the `reset` predicate is called, all TIS variables are unset (i.e. they are set to NIL), and the TIS variable is set to an empty object of the specified TIS type (see Sectionsec:specifying-tis). To set the TIS to its appropriate intial state, the `reset` predicate executes a list of TIS operations specified by the predicate `reset_operations/1`, whose argument is a set of TIS operations. This predicate must be specified in the system configuration file. An example is show in (41).

(41) (in system configuration file:)

```
reset_operations( [ set( program_state, run),
                    set( database, database_travel ),
                    set( domain, domain_travel ),
                    pushRec(private^agenda,raise(X^(task=X))),
                    pushRec(private^agenda,greet)])
```

The reset operations usually include operations to set the resource variables, thereby connecting the appropriate resources to the TIS.

## 4.8 File search paths

The `search_paths.pl` file must specify the file search path where the TrindiKit and the system itself is found. This is done using using the `assertz/1` predicate.

(42) (in `search_paths.pl` in the system directory:)

```
:- 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('<trindikitpath>/Printers'))).
:- assertz(user:library_directory(home('<systempath>'))).
```

Note that the TrindiKit distribution also contains a file called `search_paths.pl` which defines search paths for TrindiKit, AE and OAA. The installation instructions on the TrindiKit website[2] contains information about the TrindiKit `search_paths` file and how to edit it.

[2] `www.ling.gu.se/research/projects/trindi/trindikit/`

## 4.9 Additional concurrency settings

When running a system asynchronously, additional specifications must be provided by in the file `concurrent.pl`.

### 4.9.1 Agent protocol

`agent_protocol(`$P$`)`
$P$ is either `ae` or `oaa`. This defines which agent communciation protocol to use.

### 4.9.2 Settings for agent windows

- `columns(`$N$`)`
  $N$ is an integer detemining the number of columns of agent windows on the screen. 2 or 3 is recommended.
- `window_dimension_pixels(` $Width$, $Height$ `)`
  $Width$ and $Height$ are integers determining the dimensions of agents windows in pixels.
- `window_dimension_chars(` $Width$`,` $Height$ `)`$Width$ and $Height$ are integers determining the dimensions of agents windows in characters.

### 4.9.3 Display type

`display_type(` ?$Agent$`,` +$Type$ `)`
Specifies whether a certain agent should run in a separate window ($Type$=`window`) or in the background ($Type$=`background`). $Agent$ is either`tis`, `console`, `spy`, `print`, `resource` or `control`.

### 4.9.4 Resource type

`resource_type(` ?$Resource$`,` +$Type$ `)`

Specifies whether a resource *Resource* should be loaded into the TIS (*Type*=`tis`), loaded into each module that uses it (*Type*=`modules`) or run as an agent of its own (*Type*=`standalone`).

### 4.9.5 Agent language

`lang(` *Agent*`,` *Language* `)`
Specifies the programming language of an agent *Agent*. *Language* is either `sicstus` (SICStus Prolog) or `java`[3].

### 4.9.6 Showing agent communcations

`show_com(` *Agent*`,` *YesOrNo* `)`
Specify whether communication to and from agents should be output.

## 4.10 The start file

The start file contains flag settings and defines the `run/0` predicate. It also loads search paths and imports the TrindiKit `starter` Prolog module, and intialises the system.

### 4.10.1 Loading search paths and starter

The `starter` Prolog module exports the predicate `start/1` which takes the system specification filename (minus the `.pl` affix) as argument.

(43) (first in `start.pl`):

```
:- ensure_loaded( search_paths ).
:- use_module( library(starter) ).
```

[3]OAA agents can be written in Java. Consult the OAA literature for further information.

### 4.10.2 Initialization

To initialise a system according to the settings in the system configuration file (*SystemName*`.pl`), `init(` *SystemName* `)` is called. For example, the for a system `godis` the start file would contain the initialization call in (44).

(44) (second in `start.pl`):

```
:- init( godis ).
```

### 4.10.3 Setting flags

To set the flags to values other than the default values, `setflag/2` is used. This can only be done after the TrindiKit has been loaded.

(45) (third in `start.pl`:)

```
:- setflag(version,standalone).
:- setflag(show_rules,yes).
:- setflag(show_state,all).
```

Note: In serial mode, it is possible to halt a system, reset a flag, and start the system from the top. In concurrent mode, this is not advisable as it may cause TrindiKit to malfunction.

### 4.10.4 The `run` predicate

It may be convenient (though not necessary) to define a `run` predicate which may set flags before calling `start_trindikit`.

(46) (in system configuration file:)

```
run( Domain-Language ):-
        setflag(domain, Domain),
        setflag(language, Language),
        start_trindikit.
```

## 4.11 Running your system

To run a system, consult the `start.pl` file and call `start_trindikit/1` or `run/0` if it is defined. For example, to run the GoDiS system, consult the file `GoDiS2.0/start.pl` and type `run(travel-english)` to run GoDiS using the travel agency domain and English lexicon.

(47)
```
enterprise> sicstus
SICStus 3.8.4 (sparc-solaris-5.7): Mon Jun 12 18:41:59 MET DST 2000
Licensed to ling.gu.se
| ?- [start].
{consulting /users/ling/sl/Jobb/TRINDI/GoDiS/GoDiS2.0/start.pl...}
...

yes
| ?- run(travel-english).
```

## 4.12 Stopping your system

In serial mode, the system is stopped by pressing Ctrl-C and choosing (a)bort.

In asynchronous mode, the system is stopped by typing `stop` in the **console** window.

# 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.

Ljunglöf, P. (2000). Formalizing the dialogue move engine. In *Proceedings of Götalog 2000 workshop on semantics and pragma tics of dialogue*.

Martin, D. L., Cheyer, A. J., and Moran, D. B. (1999). The open agent architecture: A framework for building distributed softwar e systems. *Applied Artificial Intelligence*, **13**(1-2), 91–128.

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.

# Appendix A

# Datatype definitions

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. Below is a selection of datatypes[1] which are either included in the TRINDIKIT or will be included in future releases. Things which are not implemented in version 2.0 are marked with '*'.

In the definitions, $X$ will refer to the object to which the condition or operation is applied. For example, in the definition of the operation **add(**$El$**)** for sets, $X$ is the set to which $El$ is added. In operation definitions, $X$ can be thought of as a mutable object.

### Association set

TYPE: ASSOCSET($ElT$,$AT$)

DESC: *association set*

COND:
- **assoc(**$El$**,**$A$**)** : $El$ is associated with $A$
- **in(**$El$**)** : $El$ is a member of the set $X$, i.e. $(El, A)$ is unifiable with an pair in $X$, for some $A$
- **contains(**$El$**)** : same as in($El$)
- **empty** : $X$ is empty

OP:
- **add(**$El$**,**$A$**)** : Add $El$ to $X$ and associate it with $A$; if $El$ is already a member, do nothing
- **set_assoc(** $El$**,** $A$ **)** : associate $El$ with $A$; if $El$ is not in $X$, do nothing
- **del(**$El$**,**$A$**)** : delete an element unifiable with ($El$,$A$) from $X$
- **extend(**$S$**)** : extend $X$ with another association set $S$

[1] 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.

An association set of type ASSOC$(T_1, T_2)$ is a set where each element of type $T_1$ is associated with some other object of type $T_2$. Each element can be associated with only one object. Association sets are implemented as sets of pairs of type ($T_1$,$T_2$) where the first element of each pair is said to be associated with the second element of the pair.

### Atom

TYPE: ATOM
DESC: *Prolog atom*
COND: {
OP: {

### Bool

Objects: **true** and **false**.

TYPE: BOOL
DESC: *Boolean vale*
COND: {
OP: { **toggle** : if $X$ is **true**, set to **false**; if $X$ is **false**, set to **true**

### Counter

TYPE: COUNTER
DESC: *an integer counter*
COND: {
OP:
- **increase** : increase $X$ by 1
- **decrease** : decrease $X$ by 1
- **reset** : set $X$ to 0

### Integer

TYPE: INTEGER
DESC: *integer*

COND: $\begin{cases} ==(B): & X \text{ is identical to } B \\ <(B): & X \text{ is less than } B \\ >(B): & X \text{ is greater than } B \\ =<(B): & X \text{ is less than or equal to } B \\ >=(B): & X \text{ is greater than or equal to } B \end{cases}$

OP: $\begin{cases} +(B): & \text{add } B \text{ to } X \\ \text{-}(B): & \text{subtract } B \text{ from } X \\ *(B): & \text{multiply } X \text{ by } B \\ /(B): & \text{divide } X \text{ by } B \end{cases}$

In calls, the relational conditions and operations can optionally be written as infixes, e.g. $A < B$ instead of $< (A, B)$ or $A$: $<(B)$.

### Move

TYPE: MOVE
DESC: *dialogue move*
COND: {
OP: {

This definition does not specify any conditions, operations or objects for the MOVE type; this is left to the user.

### Participant

TYPE: PARTICIPANT
DESC: *dialogue participant*
COND: {
OP: {

Predefined objects: **usr**, **sys**.

### Program state

TYPE: PROGRAM STATE
DESC: *program state*
COND: {
OP: {

Predefined objects: **run**, **quit**.

### Queue

TYPE: QUEUE
DESC: *LIFO queue*
COND: { **empty** : $X$ is empty
OP:
- **push**($El$) : push $El$ onto $X$
- **last**($El$) : if $El$ unifies with the last element of $X$, remove it and instantiate $El$
- **extend**($Q$) : extend $X$ with another queue $Q$ by adjoining the first element of $Q$ with the last element of $X$

### Real

TYPE: REAL
DESC: *real number*
COND:
- ==($B$) : $X$ is identical to $B$
- <($B$) : $X$ is less than $B$
- >($B$) : $X$ is greater than $B$
- =<($B$) : $X$ is less than or equal to $B$
- >=($B$) : $X$ is greater than or equal to $B$

OP:
- +($B$) : add $B$ to $X$
- -($B$) : subtract $B$ from $X$
- *($B$) : multiply $X$ by $B$
- /($B$) : divide $X$ by $B$

In calls, the relational conditions and operations can optionally be written as infixes, e.g. $A < B$ instead of $<(A, B)$ or $A$: $<(B)$.

### 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(**$PathA$**,**$PathB$**)** :

For records, which can embed other complex data structures, the condition or operation "functor" $C$ may be extended with a "pseudo-type" marker prepended by "`#`", e.g. `in#rec` where the functor is `in` and the psudo-type marker is `rec`, indicating a record[2]. In this case, the first argument will be a path in leading to the embedded complex data structure which is checked by the condition. For example, `in#rec(shared.bel,` $P$`)` will check that $P$ is in the set found at the end of the path `shared.bel` in the IS.[3].

### Set

TYPE: SET($T$)

DESC: *unordered set of elements of type T*

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
- **extend(**$S$**)** : extend the set with another set $S$
- **intersect(**$S$**)** : intersect the set with $S$
- **subtract(**$S$**)** : subtract $S$ from the set

[2] A proper record type must include types for all its values, whereas the "pseudo-type" `rec` merely indicates a record of any record type.

[3] TRINDIKIT 2.0 also supports the format where $C$`#`$Type$ is replaced by $CType$, e.g. `inRec(shared.bel, A)`. However, future versions may not support this format.

### 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
- **extend**(*Stack*) : extends the stack by putting *Stack on top* of it

### 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
- **extend**(*StackSet*) : extends the stackset by putting *StackSet on top* of it
- ***del_strict**($E$) : delete the element identical with $E$ from the set
- ***del_all**($E$) : delete all elements

A stackset is a stack with some set-like porperties. Elements which are not on top of a stackset can be accessed (deleted or checked for membership). Stackset could also be called "open stacks".

### String

TYPE: STRING
DESC: *Prolog string*
COND: {
OP:
- **append**(*String*) : appends *String* to the end of $X$

# Appendix B

# Modules included in the TrindiKit package

## B.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.

(48)
```
input :-
        check_condition( input $= _ ),
        flag( input_prompt, Prompt ),
        prompt( OldPrompt, Prompt ),
        read_string( Str ),
        prompt( _, OldPrompt ),
        apply_operation( set( input, Str ) ).
```

## B.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.

(49)
```
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 ) ).
```

## B.3 A simple interpretation module

The interpretation module `Modules/interpret_simple1.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.

(50)
```
/*-------------------------------------------------------------------
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.

## B.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.

(51)
```
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.

## B.5 The "empty" interpretation and generation modules

It is possible to run TrindiKit systems without any grammar or lexicon, by typing lists of dialogue moves directly. To do this, you need to select the modules `interpret_empty` and `generate_empty`. This may be helpful for debugging before starting work on the lexicon. A sample conversation using these modules in GoDiS adapted for the mobile phone domain is shown in (52).

(52)
```
| ?- run.
$S> [greet]
$U> [greet].
$S> [ask([task(phonebook),task(messages)])]
$U> [answer(task(phonebook))].
$S> [ask([task(search_phonebook),task(add_new_number)])]
$U> [answer(task(search_phonebook))].
$S> [ask(_192560^name(_192560))]
$U> [answer(name(pelle))].
$S> [ask(call)]
$U> [answer(yes)].
$S> [inform(call_name(pelle))]
$U> [acknowledge].
$S> [ask([task(search_phonebook),task(add_new_number)])]
$U>
```

# Appendix C

# TrindiKit files

The TrindiKit is implemented by a number of Prolog files, which are listed below with short descriptions[1].

- `Agents`: agents used in asynchronous mode
  - `Agents/control.pl`: Control agent (runs sub-algorithms)
  - `Agents/tis.pl`: TIS agent
  - `Agents/print.pl`: Print agent (outputs rules and states)
  - `Agents/resource.pl` Resource agent (for stand-alone resources)
  - `Agents/spy.pl` Spy agent (for debugging)
  - `Agents/console.pl` Console agent (for debugging and quitting)
- `Datatypes`: directory of datatype definitions
- `Modules`: directory of provided module definitions
- `Printers`: directory of printer format definitions for datatypes
- `agent.pl`: TrindiKit agent library
- `agent_ae.pl`: AE specific TrindiKit agent library
- `agent_oaa.pl`:: OAA specific TrindiKit agent library
- `control_adl_parallel.pl`: Control ADL interpreter for parallel mode (used by the control agent)

[1]This list is incomplete.

- `control_adl_serial.pl`: the Control-ADL interpreter
- `control_operators.pl`: operators used in Control-ADL
- `datatypes.pl`: loads datatypes
- `dme_adl.pl`: the DME-ADL interpreter
- `error.pl`: error reporting functionality
- `find_file.pl`: library for locating files
- `flags.pl`: definitions of flags
- `infix_basic.pl`: defined operators
- `inoutput.pl`: basic input/output predicates
- `oaag.pl`: an experimental implementation of a resource accessing the OAA facilitator
- `search_paths.pl`: search paths for TrindiKit, TrindiKit modules, TrindiKit datatypes, TrindiKit printers, AE libraries (optional), OAA libraries (optional)
- `starter.pl`: TrindiKit starter library
- `start_serial.pl`: serial starter library
- `start_parallel.pl`: concurrent starter library
- `start_parallel_ae.pl`: AE-specific concurrent starter library
- `start_parallel_oaa.pl`: OAA-specific concurrent starter library
- `tis.pl`: implements check, query and update syntax
- `tis_access.pl`: Access to the TIS (used by modules)
- `tis_access_parallel.pl`: TIS access, concurrent version
- `tis_access_serial.pl`: TIS access, serial version
- `tis_operators.pl`: operators used in checks, queries and updates
- `trindikit.pl`: load trindikit

# AE files

- `ae/ae_Agent.pl`: AE agent library
- `ae/ae_AgentStarter.pl`: AE agent starter library
- `ae/ae.pl`: AE library (used by both AE agent and AE agent starter)
- `ae/ae_Com.pl`: AE interprocess communication library

# Appendix D

# The Web-demo extension

In this section, we will show how to make a Web-demo for a dialogue system implemented with the TrindiKit.

## D.1 System requirements

We have tested this on a SUN Solaris system with a Roxen webserver. With some changes (minor or major) it will probably work on other UNIX-platforms running other webservers. The client works best with Internet Explorer - the foldin/foldout features will not work in Netscape. The demo server uses CGI-scripts written in Gawk and Perl.

## D.2 Installation

The gzipped tar-archive consists of the files:

- `idemo.dat`: The installation script
- `install.txt`: Short installation description
- `Makefile`: The makefile is used by make
- `paths.txt`: A file with information about paths

- `flags.txt`: The flags that should be visible in the demo interface. The format is the same as the output from "flags." in TrindiKit.
- `allt.dat`: The data file with all scripts and html-files included in the demo

The first thing to do is to edit the file "paths.txt". It has to contain these six fields:

- `www-server` Webb address for the webbserver
- `name` Name of the demo
- `www-home` Path for the html-files
- `cgi-home` Path for the cgi-scripts
- `temp` Path for temporary files. It has to be writable and readable for the webbserver
- `system` Complete path for the Prolog file starting the dialogue system

Lines starting with # are comment lines.

Edit the `flags.txt` file. It may be empty.

Run the installer with `make`

The installer will prompt the user for a lot of paths. Just press Enter/Return if the suggested value is ok, otherwise type the correct one. It will also inform about all created files and executed commands.

### D.2.1 Technical details

The installer package consists of one gawk-script (`idemo.dat`) started by the makefile. This script will look for a lot of things in the system and build all files needed from the file `allt.dat`. The expert may edit these files to get it working on other platforms with other webservers.

## D.3 Using the demo

When the user clicks on the Start-button a CGI-script will start a Prolog server running the demo in the background and then other CGI-scripts will update the dialogue window for each utterance and create the html-files for each information state.

As mentioned before the demo works best with Internet Explorer. The screenshot shows the dialogue system GoDiS 1.2 after a few utterances. The user has clicked on the last `State` in the right window and this information state is viewed in the lower left window. The user may click on the underlined labels in the information state to hide the corresponding parts (`tmp` and `lu` are hidden in the example). The rules in the right window are hidden as a default but a click on the `Rules` tags will show (or hide) the set of rules.

### D.3.1 Stop the demo!

Always use the stop-button after running the demo. Otherwise the server process will run on the server until someone kills it. If this happens, there is a checkbox: `Stop old demos` which will kill all processes started by the demo - even other sessions than current.