Manual for GoDiS
DRAFT

Staffan Larsson

March 31, 2005

Contents

This manual is a work in progress.

Chapter 1

Introduction

This document is intended as a manual for depeloping applications for GoDiS, an Issue-based dialogue system. For an explanation of the issue-based theory and its implementation in GoDiS, see **?**.

Chapter 2

Getting started

First, you need to download and install TrindiKit and GoDiS. See Appendix **??** for instructions. Optionally, you may also want to install OAA and any speech recognition and/or TTS engine that is installed on your computer[1]. It is also very useful to have a text editor such as Emacs installed.

2.1 Running an existing GoDiS application

To run the GoDiS VCR application using text input and output, open the file `start-vcr-text.pl` located in you GoDiS directory under `godis-apps/domain_vcr` and consult it. When the prolog prompt reappears, type "`run.`" and press return. When the user input prompt (`$U`>) appears, you may type e.g. "`add a program`" (without quotes). Top the dialogue e.g. by typing "`bye`". If you don't want to see the rules and information states, stop the dialogue and type "`quiet.`". To see the rules again, type "`verb.`" (for "verbose mode).

2.2 Copying and modifying an existing GoDiS application

A good way to get started on your own application is to copy an existing application directory and successively replacing the application components.

[1]Note that you need to rebuild your TrindiKit installation after you have installed OAA, Nuance v8.0 or Nuance Vocalizer.

Chapter 3

Elements of the information-state approach to dialogue management

In this section, we briefly outline the information state approach to dialogue management. We also explain the relation between toolkits, dialogue systems, and dialogue system applications. For a more in-depth presentation of the information state apporach, see **?**or **?**.

3.1 The information state approach

The basic idea behind the "information state approach" is fairly simple. To begin with, we regard dialogue as a kind of game, where certain *dialogue moves* are possible. A common kind of dialogue move is a verbal utterance. Each move in a dialogue has effects on some kind of state containing information, and each new move is selected based on such a state. This is the basic idea, and it can be made more concrete, e.g., by considering the information state as (a part of) a *mental* state of some agent.

Dialogue is not, however, merely talking; it is also *thinking*. Thinking can also, at least to some extent, be regarded as successive updates to an information state. To model thinking in this way, we use information state *update rules* which have the form of conditionals: if x holds of the current state, then modify the state by applying operation y. For example, one could implement a rule saying that "if p is in the current state, and $p \to q$ is also in the current state, then add q to the state". This amounts to a forward-chanining modus ponens rule. In keeping with the "dialogue as game" terminology, we can regard update rules as "silent moves".

We make one cruicial, and perhaps not obvious, assumption about the relation between dialogue moves and update rules. *The relation between dialogue moves and information states can be captured completely by update rules.* That is, we don't have any representation of the preconditions and effects of dialogue moves beyond what is given in the update rules. The effects of a dialogue move M will typically be represented in a rule which has as a precondition that M was performed recently, and has not yet been integrated.

Given what we have said so far, the information state is essentially a "blackboard" structure [REF to hearsay II etc]. However, we assume that the information state is not just an unstructured jumble of information; it is an `object` of a certain `datatype`. In object-oriented programming, one defines classes of objects, and for each class a set of methods. A datatype is similar to a class; however, when definining datatypes we make a distinction between operations (which modify objects of that type) and relations, functions and selectors (which do not modify objects).

To sum up, the information state update approach to dialogue management views utterances as dialogue moves which update, and are selected on the basis of, a structured information state by means of update rules. As such, this approach is fairly general and allows the implementation of many different theories of dialogue.

3.2 TrindiKit and the information state approach

How does TRINDIKIT relate to all this? Actually, TRINDIKIT currently makes some assumptions not inherent in the information state update appoach. For one thing, it requires that modules in a dialogue system do not communicate directly with one another, but *only via the information state.* The purpose of this is to some extent ideological; we belileve it is generally a good idea to have all the information processed in the system visible, both to other modules and to the designer of the system. If we force all communication to go via the information state, we guarantee that by looking and the successive updates to the information state we can completely capture all the interactions between the modules of our system. There are also other advantages; for example, keeping all information in the IS means that the modules don't have to have any information about each other; all they need to know is ehere to read and where to write in the IS. This, in effect, means that one module can be exchanged for another without causing disturbances or requiring modifications in other modules. Unfortunately, this strategy of keeping all information in the IS has one major disadvantage: the IS tends to become a bottleneck, slowing down the system. There are various ways of improving the situation, which we will not go into here; suffice to say that we are considering lifting this limitation in future releases of TrindiKit.

An additional requirement posed by TrindiKit is that the algorithms controlling the internal distribution of work in the system (i.e., when each module should start and stop working) is separate from algorithms for updating the information state and selecting dialogue moves. This simply makes sense to us as a design principle, and again the argument is modularity. Doing things this way allows us to modify the control algorithm of a dialogue system independently of any other components.

Are there any theories which cannot be implemented using this approach? Well, a basic requirement is of course that the theory is or can be formalized

3.3 Toolkits, dialogue systems and applications

3.4 Genre-specific systems: GoDiS-IOD and GoDiS-AOD

Chapter 4

Elements of Issue-based dialogue management

This section is intended to give a brief overview of theGoDiS system and how it works.

4.1 Total Information State

4.1.1 Information state proper

The Information State (IS) is the main component of the Total Information State (TIS).

There is a basic division of IS into a record of information private to the system, and a record representing shared information.

Agenda The field /PRIVATE/AGENDA is of type Stack(Action). In general, we try to use datastructures which are as simple as possible; a stack is the simplest ordered structure so it is used as a default datastructure where order is needed as long as it is sufficient for the purposes at hand. The agenda is read by the selection rules to determine the next dialogue move to be performed by the system.

$$
\left[\begin{array}{lll}
\textsc{private} & : & \left[\begin{array}{lll}
\textsc{agenda} & : & \text{OpenQueue(Action)} \\
\textsc{plan} & : & \text{OpenStack(PlanConstruct)} \\
\textsc{bel} & : & \text{Set(Prop)} \\
\textsc{tmp} & : & \left[\begin{array}{lll} \textsc{usr} & : & Tmp \\ \textsc{sys} & : & Tmp \end{array}\right] \\
\textsc{nim} & : & \text{OpenQueue(Pair(DP, Move))}
\end{array}\right] \\
\textsc{shared} & : & \left[\begin{array}{lll}
\textsc{com} & : & \text{Set(Prop)} \\
\textsc{issues} & : & \text{OpenStack(Question)} \\
\textsc{actions} & : & \text{OpenStack(Action)} \\
\textsc{qud} & : & \text{OpenStack(Question)} \\
\textsc{pm} & : & \text{OpenQueue(Move)} \\
\textsc{lu} & : & \left[\begin{array}{lll} \textsc{speaker} & : & \text{Participant} \\ \textsc{moves} & : & \text{Set(Move)} \end{array}\right]
\end{array}\right]
\end{array}\right]
$$

$$
Tmp = \left[\begin{array}{lll}
\textsc{com} & : & \text{Set(Prop)} \\
\textsc{issues} & : & \text{OpenStack(Question)} \\
\textsc{actions} & : & \text{OpenStack(Action)} \\
\textsc{qud} & : & \text{OpenStack(Question)} \\
\textsc{agenda} & : & \text{OpenQueue(Action)} \\
\textsc{plan} & : & \text{OpenStack(PlanConstruct)}
\end{array}\right]
$$

Figure 4.1: GoDiS-AOD Information State type

Plan The /PRIVATE/PLAN is a stack of plan constructs. Some of the update rules for managing the plan have the form of rewrite rules which process complex plan constructs until some action is topmost on the plan. Other rules execute this action in case it is a system action or move it to the agenda in case it is a move-related action.

Private beliefs In GoDiS, the field /PRIVATE/BEL, a set of propositions, is used to store the results of database searches. Of course, the database (and the domain knowledge, and the lexicon) can be seen as a part of the system's private belief set, but in /PRIVATE/BEL we choose to represent only propositions which are directly relevant to the task at hand and which are the result of database searches. This is similar to seeing the database as a set of implicit beliefs, and database consultation as an inference process where implicit beliefs are made explicit. The reason for using a set is that a set is the simplest unordered datastructure.

Questions Under Discussion In GoDiS we define Questions Under Discussion, or QUD, to be an open stack of questions that can be addressed using short answers. The open stack has some set-like properties, but also retains a stack structure in case it should be useful for ellipsis resolution[1].

Issues The field ISSUES contains all questions which have been raised in a dialogue (explicitly or implicitly) but not yet resolved. It thus contains a collection of current, or "live" issues. A suitable data structure appears to be an open stack, i.e., a stack where non-topmost elements can be accessed. This allows a non-rigid modelling of current issues and task-related dialogue structure.

Actions The only addition from IOD to AOD is the ACTIONS field which has been added to /SHARED and /PRIVATE/TMP. We assume the actions stack is an open stack, which is the same structure that we use for ISSUES.

Shared Committments The field /SHARED/COM contains the set of propositions that the user and the system have mutually agreed to during the dialogue. They need not actually be believed by either participant; the important thing is that the DPs have committed to these propositions, even if only for the purposes of the conversation.

To reflect that the contents need not be true, or even privately believed by the DPS, and because we are not using situation semantics (where there is a distinction between facts and propositions) we use the label "commitments" or "committed propositions", abbreviated as COM, instead of FACTS. These, then, are propositions to which the DPs are (taken to be) jointly committed.

Latest utterance In /SHARED/LU we represent information about the latest utterance: the speaker, and the move realized by the utterance. We assume for the moment that each utterance can realize only one move. This assumption will be removed in the next chapter.

TMP

Temporary store To enable the system to backtrack if an optimistic assumption turns out to be mistaken, relevant parts of the information state is kept in /PRIVATE/TMP. The QUD and COM fields may change when integrating an ask or answer move, respectively. The plan may also be modified, e.g., if a raise action is selected. Finally, if any actions are on the agenda when selection starts

[1]Note that this is different from the way Ginzburg ?defines QUD, i.e. as containing questions which have been raised but not yet resolved, and thus currently under discussion. For reasons given in chapter 5 of ?, we have divided Ginzburgs QUD into two structures: QUD and Issues.

(which means they were put there during by the update module), these may have been removed during the move selection process.

Non-integrated moves Since several moves can be performed per turn, GoDiS needs some way of keeping track of which moves have been interpreted. This is done by putting all moves in LATEST_MOVES in a queue structure called NIM, for Non-Integrated Moves. This structure is private, since it is an internal matter for the system how many moves have been integrated so far. Once a move is assumed to be grounded on the understanding level the move is added to the /SHARED/LU/MOVES set. Since the move has now been understood on the pragmatic level, the content of the move will be a question or a full proposition (for short answers, the proposition resulting from combining it with a question on QUD).

Previous moves To be able to detect irrelevant followups, GoDiS needs to know what moves were performed (and grounded) in the previous utterance. These are stored in the /SHARED/PM field.

4.1.2 Module interface variables

4.1.3 Resource interface variables

4.2 Dialogue moves

While dialogue move types are often defined in terms of sentence mood, speaker intentions, and/or discourse relations (see e.g. **?**), we opt for a different solution. In our approach, the type of move realized by an utterance is determined by the relation between the content of the utterance, and the activity in which the utterance occurs.

4.2.1 Core dialogue moves in GoDiS-IOD

The following dialogue moves are used in GoDiS:

- ask(q), where q : Question
- answer(a), where a : ShortAns or a : Proposition

- greet

- quit

In inquiry-oriented dialogue, the central dialogue moves concern raising and addressing issues. This is done by the ask and answer moves, respectively. The greet and quit moves are used in the beginning and end of dialogues to greet the user and indicate that the dialogue is over, respectively.

4.2.2 Grounding moves

4.2.3 Additional moves in GoDiS-AOD

In addition to the dialogue moves listed above, GoDiS-AOD uses the following two moves:

- request(α), where α : Action

- confirm(α), where α : Action

These two moves are sufficient for activities where actions are performed instantly or near-instantly, and always succeed. If these requirements are not fulfilled, the confirm move can be replaced by or complemented with a more general report(α, $Status$) move which reports on the status of action α. Possible values of $Status$ could be done, failed, pending, initiated etc.; report(α, done) would correspond to confirm(α).

4.3 Dialogue Move Engine

4.3.1 Update module

Move integration

Grounding

Question accommodation

Plan execution

Downdating QUD, ISSUES and ACTIONS

4.3.2 Selection module

Action selection

Move selection

4.4 Resource interface

The resource interface can be seen as a mediator between (domain-independent) modules and (domain-specific) resources. The resource interface makes it possible to check and update resources is update rules. Since the TIS is abstract datastructure containing objects of various types, it makes sense that resources that are connected to the TIS should also be regarded as objects of certain datatypes.

A straighforward solution to this would be to let each resource contain a definition of the type of that resource, i.e. a list of relations, functions, selectors and operations that can be applied to the resource. However, it turns out that this leads to a lot of redundancy in the code since each resource object of a certain type has to define the same type. Instead, the definitions of resource types reside in the *resource interface* file. The resource object merely exports a number of predicates in the normal Prolog way. The resource interface specification uses these predicates to define the relations etc. that determine how the resource can be accessed from the modules.

4.5 Dialogue plans

In this section, we introduce a formalism for representing procedural plans as sequences of actions. Dialogue plans are implemented in the domain resource (see Section ??).

4.5.1 Action sequences and actions

In the simplest case, a plan consists of a sequence of actions. More complex plans may also include e.g. conditionals (if-then-else). In general, dialogue plans are built from so-called plan constructors.

Action sequences have the form $\langle a_1, \ldots, a_n \rangle$ where a_i **:** Action $(1 \leq i \leq n)$.

All dialogue moves are actions, which means they can be included in dialogue plans. There are also more abstract actions which however are connected to dialogue moves. Third, there are actions for manipulating the information state in various ways. Third, there are actions for accessing resources, e.g. actions enabling database consultation and interaction with devices.

In the following, q is a question, p is a proposition, and a is an action or action sequence.

4.5.2 Actions connected to dialogue moves

In GoDiS, there are three action types closely related to dialogue move types.

findout(q) Find the answer to q. This is typically done by asking a question to the user, i.e., by performing an ask move, and hoping for an answer. The findout action is not removed until the question q has been publically resolved. A question is publically resolved when a resolving answer to the question is in the set of jointly comitted propositions (i.e. /SHARED/COM)[2]

[2] Although they have not yet been implemented in GoDiS, there are also in principle other ways of achieving public resolvedness, e.g. that the system finds the information by consulting some resource or by inferring from its own private beliefs, and then provides this information to the user. Note that it is necessary that the answer is communicated to the user for the findout action to be complete.

raise(q) Raise the question q. This action is similar to findout, except that it is removed from the plan as soon as the ask-move is selected. This means that if the user does not answer the question when it has been raised, it will not be raised again. This is useful e.g., for requesting optional information.

respond(q) Provide an answer to question q. This is done by performing an answer move with content p, where p is a resolving answer to q.

Actions for manipulating the information state

bind(q)

protect(q)

forget(p)

forget_all

forget_except(p

assume(p)

assume_issue(q)

4.5.3 Resource-related actions

consultDB(q) For a typical information-exchange task, the application is a static database[3].

The consultDB(q) action (where q is a question) which will trigger a database search which takes all the propositions in /SHARED/COM and given this looks

[3]Note that "static" here does not mean that the database cannot be updated. It only means that it is not updated by the dialogue system.

up the answer to q in the database. The resulting proposition is stored in /PRIVATE/BEL.

dev_do(dev, α**)**

dev_set(dev, var, val**)**

dev_get(dev, var**)**

dev_query(dev, q**)**

dev_queryAll(dev, q**)**

change_domain

change_language

Conditionals

if_then(p**,** a**)** If p is in /PRIVATE/BEL or /SHARED/COM, then replace if_then(p, a) with a; otherwise, delete it.

if_then_else(p**,** a_1**,** a_2**)** If p is in /PRIVATE/BEL or /SHARED/COM, then replace if_then(p, a_1, a_2) with a_1; otherwise, replace it by a_2.

4.5.4 Some example plans

(4.1) ISSUE : $?x.\mathbf{price}(x)$
PLAN: ⟨
findout($?x.\mathbf{means\_of\_transport}(x)$),
findout($?x.\mathbf{dest\_city}(x)$),
findout($?x.\mathbf{depart\text{-}city}(x)$),
findout($?x.\mathbf{depart\text{-}month}(x)$),
findout($?x.\mathbf{depart\text{-}day}(x)$),
findout($?x.\mathbf{class}(x)$),
findout(**?return**),
if_then(**return**, ⟨ findout($?x.\mathbf{return\text{-}month}(x)$),
findout($?x.\mathbf{return\text{-}day}(x)$) ⟩),
consultDB($?x.\mathbf{price}(x)$)
⟩

(4.2) a. ACTION : vcr_add_program
PLAN: ⟨
findout($?x.\mathbf{channel\_to\_store}(x)$)
findout($?x.\mathbf{date\_to\_store}(x)$)
findout($?x.\mathbf{start\_time\_to\_store}(x)$)
findout($?x.\mathbf{stop\_time\_to\_store}(x)$)
dev_do(vcr, 'AddProgram')
⟩
POST : **done('AddProgram')**

4.6 Formal semantic representations

Here we describe the syntax of the simple formal semantic representation currently used in GoDiS[4]. This description defines a set of content types which are explained and exemplified below. The symbol ":" represents the of-type relation, i.e., *Expr* **:** *Type* means that *Expr* is of type *Type*.

Atom types

Pred$_n$, where $n = 0$ or $n = 1$: n-place predicates, e.g., **dest-city**, **month**
Ind: Individual constants, e.g., **paris**, **april**
Var: Variables, e.g., $x, y, \ldots, Q, P, \ldots$

[4]It should be noted that the GoDiS DME is independent semantic formalism, as long as the appropriate semantic properties and relations are defined for an application. To implement a GoDiS application with a different kind of semantics, the resource interface definitions (`resource_interfaces.pl`) need to be modified.

Sentences

$Expr$: Sentence iff $Expr$: Proposition or $Expr$: Question or $Expr$: ShortAns

$Expr$: Proposition if

- $Expr$: Pred_0 or
- $Expr = pred_1(arg)$, where arg : Ind and $pred_1$: Pred_1 or
- $Expr = \neg P$, where P : Proposition or
- $Expr = \mathbf{fail}(q)$, where q : Question

$Expr$: Question if $Expr$: YNQ or $Expr$: WHQ or $Expr$: ALTQ

$?P$: YNQ if P : Proposition

$?x.pred_1(x)$: WHQ if x : Var and $pred_1$: Pred_1

$\{ynq_1, \ldots, ynq_n\}$: ALTQ if ynq_i : YNQ for all i such that $1 \leq i \leq n$

$Expr$: ShortAns if

- $Expr = \mathbf{yes}$ or
- $Expr = \mathbf{no}$ or
- $Expr$: Ind or
- $Expr = \neg arg$ where arg : Ind

4.6.1 Propositions

Propositions are represented by basic formulae of predicate logic consisting of an n-ary predicate together with constants representing its arguments, e.g., **loves(john,mary)**.

In a dialogue system operating in a domain of limited size, it is often not necessary to keep a full semantic representation of utterances. For example, a user utterance of “I want to go to Paris” could normally be represented semantically as e.g., **want(user, go-to(user, paris))** or **want(u, go-to(u,p)) & city(p) & name(p, paris) & user(u)**. GoDiS uses a *reduced* semantic representation with a coarser, domain-dependent level of granularity; for example, the above example will be rendered as **dest-city(paris)**. This reduced representation is in part a consequence of the use of keyword-spotting in interpreting utterances, but can arguably also be regarded as a reflection of the level of semantic granularity inherent in the underlying domain task. As an example of the latter, in a travel agency domain there is no point in representing the fact that it is the user (or customer) rather than the system (or clerk) who is going to Paris; it is implicitly assumed that this is always the case.

As a consequence of using reduced semantics, it will be useful to allow 0-ary predicates, e.g., **return**, meaning "the user wants a return ticket".

The advantage of this semantic representation is that the specification of domain-specific semantics becomes simpler, and that unnecessary "semantic clutter" is avoided. On the other hand, it severely restricts the possibility of providing generic semantic analyses that can be extended to other domains.

If the database search for an answer to a question q fails the resulting proposition is **fail**(q). We have chosen this representation because it provides a concise way of encoding a failure to find an answer to q in the database.

4.6.2 Questions

Three types of questions are handled by GoDiS: *y/n*-questions, *wh*-questions, and alternative questions. Here we describe how these are represented on a semantic level; the syntactic realization is defined in the lexicon.

- *y/n*-questions are propositions preceded by a question mark, e.g., **?dest-city(london)** ("Do you want to go to London?")

- *wh*-questions are lambda-abstracts of propositions, with the lambda replaced by a question mark, e.g., **?x.dest-city(x)** ("Where do you want to go?")

- alternative questions are sets of *y/n*-questions, e.g. **{?dest-city(london), ?dest-city(paris)}** ("Do you want to go to London or do you want to go to Paris?")

4.6.3 Domain ontology (semantic sortal restrictions)

GoDiS uses a rudimentary ontology consisting of domain-dependent semantic sortal categories. Sorts are useful e.g. for distinguishing non-meaningful propositions from meaningful ones. However, what is meaningful in one activity may not be meaningful in another, and vice versa. Therefore, the sortal system is implemented as a part of the domain knowledge. Another prominent use of sorts is to determine whether an answer is relevant to (about, in Ginzburg's terminology) a certain question (see Section **??**).

Individuals and sorts

All members of Ind are assigned a sort. For example, the travel agency domain includes the sorts **city**, **means_of_transport**, **class**, etc. The individual constant **paris** has sort **city** and **flight** has sort **means_of_transport**.

Sortal hierarchy

isa

Sortal correctness of propositions

The property of a proposition P being sortally correct is implemented in GoDiS as sort-restr(P). A proposition is sortally correct if its arguments fulfil the sortal constraints of the predicate. For example, the proposition $\mathbf{dest\_city}(X)$ is sortally correct if the sort of X is **city**. Sortal constraints of predicates are implemented in the domain resource, as exemplified in (4.3).

(4.3) sort_restr($\mathbf{dest\_city}(X)$) $\leftarrow$ sem_sort(X, **city**).

Chapter 5

Non-DME modules used by GoDiS

This section describes the modules supplied with TrindiKit that are used by GoDiS.

The TrindiKit package includes a couple of simple modules which can be used to quickly build prototype systems.

- **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

5.1 Simple text input module

The input module **input_simpletext** 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.

5.2 Text input with simulated recognitions score

5.3 Nuance ASR input

5.4 A simple interpretation module

The interpretation module **interpret_simple1** 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**, **means_of_transport** etc.) and synonymy sets. A synonymy set is a set of words 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 realize a greet move.:

(5.4) input_form([hello], greet)

The following rule says that a phrase consisting of the word "to" followed by a phrase S constitutes 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 synonymy set of words with the meaning C.

(5.5) input_form([to| S], answer(to(C)) $\leftarrow$ 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.

5.5 A simple generation module

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

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

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.

5.6 Simple text output module

The output module **output_simpletext** takes the string in the system variable OUTPUT and prints it on the computer screen, preceded 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.

5.6.1 Nuance Vocalizer output

Chapter 6

The components of a GoDiS application

This sections outlines the components of a GoDiS application.

6.1 File structure

It is highly advisable to keep trindikit, godis and godis applications in separate directories. This enables e.g. updating godis to a new version without touching the applications.

- trindikit
- godis/
  - godis-basic
  - godis-grounding
  - godis-iod
  - godis-aod
- godis-apps/
  - vcr-godis
  - ta-godis
  - ...

6.2 The GoDiS application specification file

This file specifes datatypes, modules and resources used by your application, as well as additional parameters.

Some applications may have variants, in which case each variant has a separate specification file (and start file). For example, an application might have one text-based variant and one speech-based variant.

MORE

6.2.1 Selecting a GoDiS variant

Depending on the properties of your application domain, there is a choice between different variants of GoDiS.

- godis-basic: multiple simultaneous tasks, information sharing between plans
- godis-grounding: as godis-basic, plus grounding
- godis-iod: as godis-grounding, plus accommodation and associated grounding strategies
- godis-aod: as godis-iod, plus facilities for action-oriented dialogue

6.3 The start file

This file contains specification of directories and files that should be consulted when running GoDiS. To load GoDiS, this file should be consulted.

When building a new application, this is where you specify which GoDiS version to use, the home directory of you application, the name of the GoDiS application specification file for your application, and any additional libraries used by the application.

6.4 Resources and the GoDiS resource interface

The methods available for a resource are not necessarily (or even usually) defined in the resources itself. Rather, the resource defines a set of prolog predicates which are utilized by the *resource interface*. This interface describes each resource as an object of some type, and for each type it defines a set of methods (relations, functions, selectors, and operations). These methods are defined in terms of the predicates exported from the resource itself.

6.5 The domain resource

6.5.1 Dialogue plans

In GoDiS, the domain resource includes a set of dialogue plans which contain information about what the system should do in order to achieve its goals. In dialogue, dialogue plans are loaded into the information state and executed by update rules, which "consume" them step by step. Each plan is associated with a goal which can either be a question or an action. If the user asks a question q, there is an update rule which looks in the domain resource for a plan for dealing with q, and if one is found, loads it into the information state.

See Sectionsec:dialogue-plans for an explanation of GoDiS dialogue plans.

Dialogue plans are mainly specified using the predicate `plan/2`, but there are also some additional predicates.

- `plan(Goal, Plan)`
- `postcond(Action, Plan)`
- `depends( Q1, Q2)`

In the domain resource, sequences of actions are represented by prolog lists.

The domain resource also specifies a domain-specific ontology which is used by the dialogue move engine to determine e.g. question-answer relations (relevance and resolvedness).

6.5.2 Sortal restrictions

GoDiS uses a rudimentary system of domain-dependent semantic sortal categories. For example, the travel agency domain includes the sorts **city**, **means_of_transport**, **class**, etc. All members of Ind are assigned a sort; for example, the individual constant **paris** has sort **city** and **flight** has sort **means_of_transport**.

Sorts make it possible to distinguish non-meaningful propositions from meaningful ones. However, what is meaningful in one activity may not be meaningful in another, and vice versa. Therefore, the sortal system is implemented as a part of the domain knowledge. In GoDiS, the sorts are mainly used for determining whether an answer is relevant to (about, in Ginzburg's terminology) a certain question (see Section **??**).

The property of a proposition P being sortally correct is implemented in GoDiS1 as sort-restr(P). A proposition is sortally correct if its arguments fulfil the sortal constraints of the predicate. For example, the proposition **dest_city**(X) is sortally correct if the sort of X is **city**. Sortal constraints of predicates are implemented in the domain resource, as exemplified in (6.7).

(6.7) sort_restr(**dest_city**(X)) $\leftarrow$ sem_sort(X, **city**).

proposition	**restriction**
dest_city(X)	$X \in$ CITY
depart_city(X)	$X \in$ CITY
how(X)	$X \in$ MEANS_OF_TRANSPORT
depart_month(X)	$X \in$ MONTH
depart_day(X)	$X \in$ DAY
class(X)	$X \in$ CLASS
price(X)	$X \in$ PRICE

Table 6.1: Sortal restrictions on propositions in the Travel Agency domain

RELATE TO PROLOG CODE

6.5.3 Sortal hierarchy

The sortal restrictions on proposition is defined in terms of a hierarchy of semantic sorts, defined by the `sem_sort`/2 relation. Since this hierarchy is also useful for the lexicon resource, it is kept in a separate file (`semsort_....pl`).

As an example, we show the sortal hierarchy from the travel agency domain:

$$\textsc{top}\left\{\begin{array}{l}
\textsc{city}\left\{\begin{array}{l}\mathsf{paris}\\ \mathsf{london}\\ \mathsf{goteborg}\\ \ldots\end{array}\right.\\
\textsc{means_of_transport}\left\{\begin{array}{l}\mathsf{plane}\\ \mathsf{boat}\\ \mathsf{train}\end{array}\right.\\
\textsc{month}\left\{\begin{array}{l}\mathsf{january}\\ \mathsf{february}\\ \ldots\end{array}\right.\\
\textsc{day}\left\{\begin{array}{l}1, 2, \ldots, 31\end{array}\right.\\
\textsc{class}\left\{\begin{array}{l}\mathsf{economy}\\ \mathsf{business}\end{array}\right.\\
\textsc{price}\left\{\begin{array}{l}\mathsf{Nat}\end{array}\right.
\end{array}\right.$$

RELATE TO PROLOG CODE

6.6 The device/database resource

6.7 The lexicon resource

6.8 The speech recognition grammar resource

Appendix A

Installation instructions

This section describes the steps necessary to download and install GoDiS and TrindiKit, the toolkit on which GoDiS is built.

A.1 Downloading and installing TrindiKit

TrindiKit can be accessed either via the TrindiKit webpage (http://www.ling.gu.se/projekt/trindi//trindikit/), via the TrindiKit SourceForge website (http://sourceforge.net/projects/trindikit/), or via anonymous CVS from sourceforge. The latter is prefereable if you want to get the absolutely latest version; however, the documentation is not always up to date and some parts of the toolkit may be incomplete. If you want a well-documented release, the first two options are preferable.

A.1.1 Prerequisites

Trindikit runs under Windows, Unix and Linux. SICStus prolog 3.8 or later is needed. In addition, you need the following:

- JDK 1.4 - needed to compile the java OAA agents and the GUI and to run the build script

- JRE 1.4 (Included in JDK) - needed to run the build script, see INSTALLATION, and to use the java OAA agents and the GUI

In addition the following software is useful but not necessary:

- Nuance ASR 8 - speech recognition software, needed to use the module `input_nuance_oaa_basic.pl`
- Nuance Vocalizer 1 - text-to-speech software, needed to use the module `output_nuance_oaa_basic.pl`
- Open Agent Architecture (OAA) 2.2 (or later) - needed to use the TRINDIKIT OAA facilities, and the modules `input_nuance_oaa_basic.pl` and `output_nuance_oaa_basic.pl`

A.1.2 Installing a CVS client

Windows For Windows, we recommend TortoiseCVS, a graphical CVS client written in Java. I can be found at www.tortoisecvs.org. Download, unzip and install.

UNIX/Linux ...

Include the following in your .rc.user.d/chs/environment file:

```
setenv CVS_RSH ssh
```

A.1.3 Downloading and unzipping TrindiKit

A.1.4 Accessing TrindiKit via anonymous CVS

Windows Right-click on the desktop, and select "CVS checkout". A dialogue box will appear, which should be filled in thus:

- Protocol: `:pserver`
- Server: `cvs.trindikit.sourceforge.net`
- Repository folder: `/cvsroot/trindikit`
- User name: `anonymous`
- Module: `trindikit`

Click "OK" and follow the instructions. If you are asked for a password, just press "Return" or click "OK".

UNIX/LINUX cvs -d :pserver:anonymous@cvs.trindikit.sourceforge.net:/cvsroot/trindikit login [supply no password if prompted] [to checkout our read-only version of trindikit] cvs -d :pserver:anonymous@cvs.trindikit.sourceforge.net:/cvsroot/trindikit checkout trindikit

A.1.5 Installing TrindiKit

Setting the TRINDIKIT environment variable

First, you need to set the TRINDIKIT environment variable.

Windows In the Start menu, Go to "settings" → "control panel" → "system" → "advanced" → "Environment variables". Under "System variables" click "New" and fill in the fields thus:

- Variable name: `TRINDIKIT`
- Variable value: here, enter the path to the "dist" directory located in your "trindikit" directory; the exact path depends on where you saved TrindiKit when downloading it. Example: `C:\MyCVS\trindikit\dist`.

UNIX/LINUX setenv TRINDIKIT ....

Running the TrindiKit build script

Windows Open a DOS window (Start → Programs → Accessories → Command interpreter), go to the `trindikit` folder (using the `cd` command, e.g. `cd C:\MyCVS\trindikit`), and type `build all`. If everything is okay, the system should report that the build was successful.

A.1.6 TrindiKit directory structure (after installation)

```
doc/api - api for the TRINDIKIT java classes
doc/manual - the trindikit manual
lib - external programs and utilities
licenses - various licenses for external programs and utilities
test - for testing
```

```
src/prolog/trindikit - "core" TRINDIKIT
src/prolog/ae - 'agent environment', used for running TRINDIKIT
   asynchronously
src/prolog/examples - example dialogue systems
src/java - the TRINDIKIT java source tree, including OAA agents and the
   GUI
dist/classes - the TRINDIKIT java class tree
dist/prolog/trindikit - "core" TRINDIKIT (with appropriate search paths
   set)
dist/prolog/ae - 'agent environment'
examples - example dialogue systems built using TRINDIKIT
examples/bin - scripts for running example systems
examples/godis - an example system built using TRINDIKIT
```

A.2 Downloading and installing GoDiS

GoDiS currently has no webpage and is only available via anonymous CVS.

A.2.1 Prerequisites

SICStus Prolog 3.8 or later.

A.2.2 Accessing GoDiS via anonymous CVS

Make sure you have installed a CVS client (see Section A.1.2).

Windows Right-click on the desktop, and select "CVS checkout". A dialogue box will appear, which should be filled in thus:

- Protocol: `:ext`
- Server: `mozart.gslt.hum.gu.se`
- Repository folder: `/users/gslt/cvs/repository`
- User name: `anonymous`
- Module: `godis`

Click “OK” and follow the instructions.

UNIX/LINUX

```
[first time you have to login]
cvs -d :pserver:anonymous@mozart.gslt.hum.gu.se:/users/gslt/cvs/repository login
[supply no password if prompted]
[to checkout our read-only version of GoDiS]
cvs -d :pserver:anonymous@mozart.gslt.hum.gu.se:/users/gslt/cvs/repository checkout godis
```

A.2.3 Installing GoDiS

Setting the GODIS environment variable

Windows In the Start menu, Go to “settings” → “control panel” → “system” → “advanced” → “Environment variables”. Under “System variables” click “New” and fill in the fields thus:

- Variable name: `GODIS`
- Variable value: here, enter the path to the “dist” directory located in your “godis” directory; the exact path depends on where you saved GoDiS when downloading it. Example: `C:\MyCVS\godis\dist`.

Running the build script

Windows Open a DOS window (Start → Programs → Accessories → Command interpreter), go to the `godis` folder (using the `cd` command, e.g. `cd C:\MyCVS\godis`), and type `build all`. If everything is okay, the system should report that the build was successful.

A.2.4 GoDiS directory structure

Appendix B

Downloading and installing additional software

B.1 Installing Java

B.1.1 Windows

RECOMMEND SOME JAVA DOWNLOAD!

Make sure that the PATH variable (under Windows) includes a path to the file `javac.exe`. If not, search for the file by selecting Start → Search → Files and folders. Then, manually edit the PATH variable as follows: In the Start menu, Go to "settings" → "control panel" → "system" → "advanced" → "Environment variables". Select "PATH" and go to the end; add ";" followed by the path to the `javac.exe` file.

B.2 Downloading and installing OAA

B.3 Installing Nuance ASR

B.3.1 Windows

B.3.2 UNIX/Linux

B.4 Installing Nuance Vocalizer

B.4.1 Windows

B.4.2 UNIX/Linux

Appendix C

Using GoDiS with Nuance v8.0 and Vocalizer

C.1 Getting Nuance and Vocalizer to run on your computer

C.1.1 Testing Nuance ASR

```
*kr igng en license-manager
        (jag tror att jag gjorde en bat-fil som hette  start_nlm el. ngt)

*ppna ett dosfnster

*g till C:\Program\Nuance\v8.0.0\sample-packages o skriv
        'recserver -package digits' (el ngt annat igenknningspaket)

*ppna ett dosfnster till

*g till samma stlle, skriv 'Xapp -package digits lm.Addresses=localhost'
```

C.1.2 Testing Vocalizer

C.2 Configuring your GoDiS application to use Nuance

C.3 Running Nuance and Vocalizer with your application

Unfortunately, running GoDiS with Nuance and Volcalizer requires going through a number of steps to get everything running.

Step 1: Compile your nuance grammars

Go to the directory where the .grammar files are and run the nuance-compile script on them.

Example:

```
nuance-compile asrg_travel_english English.America.1 -auto_pron lm.Addresses=localhost
```

Step 2: start the Nuance license manager

This is done by running the `nlm` script with your license number as argument.

Step 3: Start OAA

Go to your OAA installation, go to the `runtime` subdirectory, and run the `fac` script.

This requires that OAA is installed, and that the file `setup.pl` is directly under the `C:` directory. This is achieved by editing the file `setup1.pl` located in the OAA runtime directory, changing the text `ChangeMe` to the name of your machine.

Edit the setup-nuance script

This is a script that executes a number of commands automatically. To make it work, you need to copy the script from an existing Nuance-connected GoDiS application and change all paths in the file to match your installations. The file should be stored in the application directory.

Run setup-nuanuce

Go to your application directory, and type `setup-nuance`.

If no error messages have appeared, you can now start your GoDiS application.