Dialogue Dynamics in Restricted Dialogue Systems

Johan Bos Stina Ericsson Staffan Larsson Ian Lewin
Peter Ljunglöf Colin Matheson

Distribution: PUBLIC

Task Oriented Instructional Dialogue
LE4-8314

Deliverable D3.2

February 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. Johan Bos was the overall editor and author of Chapter 3. The system architecture described in Chapter 2 was a collaborative result of the Trindi Consortium. Chapter 4 was written by Ian Lewin, Chapter 5 by Colin Matheson, and Chapter 6 by Stina Ericsson and Staffan Larsson.

Contents

Chapter 1

Introduction

This document reports on a first evaluation of the dialogue move engine (DME) proposed in the Trindi-project (the theory of information state change, see [CLM+99]) with respect to restricted dialogue. The DME-model is implemented by the TrindiKit. For documentation of the TrindiKit the reader is referred to [LBBT99].

Evaluation of dialogue move engines is, as is well known, a difficult task [BBL+99]. The evaluation reported here addresses two issues:

1. Use of the TrindiKit
2. Evaluation against the "Tick-list"

The first issue is almost purely technical, and evaluates the use of the TrindiKit to build new or adapt existing dialogue systems. Considered will be four implementations developed in the Trindi-project using the TrindiKit: Midas (Chapter 3), the SRI Autoroute system (Chapter 4), an implementation of the Poesio-Traum-Theory of information states (EDIS, Chapter 5), and GoDiS (Chapter 6). These implementations—all based around the route planning scenario (or small extensions of it)—differ with respect to the level in which they represent a complete dialogue system. Some of them still expose a fairly theoretical character (for instance the PTT model in EDIS, Chapter 5) and presuppose external components for speech-analysis, whereas others more or less exhibit a full dialogue system (e.g., the SRI Autoroute system).

The second issue covers the examination of the "Tick-list" developed in [BBL+99]. The Tick-list is a set of yes-no questions that examiners attempt to answer by trying out certain dialogues on systems. The goal of this enterprise is, primarily, to acquire knowledge about

the systems under evaluation, and, secondarily, to extend the criteria on the Tick-list itself—the Tick-list is not a fixed point. In this report, evaluation of the Tick-list is only applied to those implementations that more or less resemble a practical system.

This report is outlined as follows. Chapter 2 describes the functionality of the TrindiKit and introduces the Tick-list. The four implementation mentioned above will be discussed in Chapters 3–6.

Chapter 2

Dialogue Move Engines and Information States

2.1 Information States and Updates

The Trindi Model presents itself as an *Information State Update* approach to dialogue. Dialogue utterances are analysed as information state transformers. Information states do not alter in arbitrary ways—they alter in ways determined, at least in part, by the types of the dialogue utterances involved. These types are commonly called *moves*. The way in which moves alter the information state is to be modelled by *update rules*. In order to participate in a dialogue, an agent must be able to understand the moves of his partners and deliberate about and construct moves of his own. The model is an extremely general one and, indeed, part of the interest in it lies in the extent to which, by casting alternative dialogue theories in it, one might be able to compare and contrast, for example, the different types of information state that they invoke. A brief outline of two different dialogue models (finite state dialogue models and classical plan-based approaches) and how they might be constructed within the Trindi model was outlined in [TBC+99].

2.2 The Dialogue Move Engine

A Dialogue Move Engine (DME) is that part of a dialogue system responsible for operations on the system's representation of the ongoing dialogue, or using Trindi terminology, the *information state*. These operations typically consist of:

1. updating the information state according to dialogue moves performed by the participants of a dialogue;
2. generating appropriate moves to be performed by the system itself.

In the Trindi project, a software package for building and experimenting with dialogue move engines has been developed (TrindiKit). Apart from proposing a general system architecture, this toolkit also specifies formats for defining information states, update rules, selection rules, dialogue moves and associated algorithms. It further provides a set of tools for experimenting with different formalizations of implementations of information states, rules, and algorithms.

To build a dialogue move engine, one needs to provide definitions of rules, moves and (optionally) 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., which can then be used as *resources* by the DME.

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. The TrindiKit also stimulates to integrate the DME-technology into existing dialogue systems (see Chapter 4 for an example).

Updating information states goes hand in hand with update rules, selection rules and dialogue moves. Their definitions are repeated here for convenience:

- Update rule: a rule used by the update module to update the information state or values of interface variables. The update rules are formulated in terms of conditions and operations on the information state.
- Selection rule: a rule used by the selection algorithm to select a move to be performed by the system. Formulated basically as a u-rule, but also specifies a (set of) dialogue move(s).
- Dialogue move: a move or act performed by user or system, associated with an utterance. Sometimes we use the term 'tacit move', which is the act of applying an update rule or selection rule to the information state, performing the operations specified by the rule.

2.3 TrindiKit

Using the TrindiKit (and its architecture) obligates the developer to the following actions:

- providing a set of dialogue moves
- providing a set of update and selection rules
- selecting an update and control algorithm
- specifying the information state according to the provided data-types

The general architecture TrindiKit is assuming is one where communication between the modules is organized via *interface variables*. For a simple DME, the following interface-variables are suggested:

- `input`: the input utterance(s)
- `latest_moves`: list of moves currently under consideration (interpretation)
- `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)

The information state can be seen as an additional interface variable. Each module, whether it be DME-external (such as parsers, generators, recognizers) or DME-internal (update algorithms, selection algorithms) should specify which interface variables they need to modify. In principle, each module can *access* any interface variable and the information state. Only the update-module (see below) can change the information state.

The components (modules) in the architecture are the following:

- Input module (utterance recognition): Receives input utterances from the user and stores it in the interface variable `input`.
- Interpretation module: Takes utterances (stored in `input`) and gives interpretations in terms of d-moves (including semantic content). The interpretation is stored in the interface variable `latest_moves`.
- Update module: Applies update rules to the information state according to the update engine algorithm.
- Selection module: Selects dialogue move(s) using the selection rules and the move selection algorithm chosen. The resulting moves are stored in the information state. The update engine and the selection engine together form the core of the dialogue move engine.

- Generator: Generates output utterances (stored in the interface variable `next_moves`) based on the contents of the information state and passes these on to utterance synthesis, via the interface variable `output`.
- Output (utterance synthesis): Produces system utterances.
- Control: wires together the other modules, either in sequence or through some asynchronous mechanism.

2.4 The Trindi Tick-list

The Trindi wish list of desired dialogue behavior is specified as a list of yes-no questions. The tick-list is expected to develop both in terms of the number and content of the questions themselves and in the range of possible answers. The questions to be ticked or crossed are listed below.

Currently the list is restricted to fifteen questions. There are a few additions to the previous Tick-list as presented in [BBL+99]. The original question **Qn 1** is split into items **Qn 2** and **Qn 3**. The tick-list items **Qn 14** – **Qn 17** are new.

Qn 1 *Is utterance interpretation sensitive to dialogue context?*

Qn 2 *Is utterance interpretation sensitive to deictic context?*

Qn 3 *Can the system deal with answers to questions that give more information than was requested?*

Qn 4 *Can the system deal with answers to questions that give different information than was actually requested?*

Qn 5 *Can the system deal with answers to questions that give less information than was actually requested?*

Qn 6 *Can the system deal with ambiguous designators?*

Qn 7 *Can the system deal with negatively specified information?*

Qn 8 *Can the system deal with no answer to a question at all?*

Qn 9 *Can the system deal with noisy input?*

Qn 10 *Can the system deal with 'help' sub-dialogues initiated by the user?*

Qn 11 *Can the system deal with 'non-help' sub-dialogues initiated by the user?*

Qn 12 *Does the system only ask appropriate follow-up questions?*

Qn 13 *Can the system deal with inconsistent information?*

Qn 14 *Can the system deal with belief revision?*

Qn 15 *Can the system deal with barge-in input?*

Qn 16 *Is it possible to get a system tutorial concerning the kind of information the system can provide and what the constraints on input are?*

Qn 17 *Does the system check its understanding of the user's utterances?*

These questions can be answered with yes-no answers, conform to a 'Black Box' assessment of a dialogue system. As the systems examined in this report are still under development, we will try to use the Tick-list criteria in a 'Glass Box' fashion, that is, explain why a criterion is fulfilled or why not.

Chapter 3

Midas

In this chapter we provide some evaluation of the TrindiKit model with respect to the MIDAS system and of MIDAS itself with respect to the Trindi Tick-list.

3.1 System Description

Midas (Multiple Inference-based Dialogue Analysis System) is a prototype of a dialogue system exploring semantic representation and first-order reasoning to model and plan the ongoing dialogue.[1]

3.1.1 The lexicon, grammar, and parsing

The lexicon contains about 2700 inflected forms, divided into 502 nouns, 438 names, 81 adjectives, 312 intransitive verbs, 1096 transitive verbs, 32 ditransitive verbs, 57 prepositions, 25 determiners, and a bunch of miscellaneous categories. The entries are based on the Autoroute corpus. For nouns, a subset of the WordNet ontology of hyper- and hyponyms [Mil95] is included in Midas' lexicon . These are used to generate "isa" and disjointness conditions upon the different objects. Cue-phrases contain dialogue-acts in their lexical entries.

[1] Midas is developed (and actually still under development) by Johan Bos at the Department of Computational Linguistics, University of the Saarland. A web-interface is available at `http://www.coli.uni-sb.de/~bos/trindi/midas.html`.

A simple straightforward phrase structure rule grammar is used. Utterances can be (multiple) sentences, noun phrases, prepositional phrases, questions (yes-no and simple wh-questions), or cue-phrases (such as greetings, "yes", "no"). On the sentence level conditionals and disjunctions are covered. There is a basic noun phrase coverage, including modification of nouns by adjectives, prepositional phrases, or relative clauses. Further, noun phrases cover proper names, mass nouns, pronouns, and possessives. Verb phrases include auxiliaries (to be and to have), modals, and to-complement verbs. Coordination is covered for basic categories such as nouns, adjectives, and verbs. Comparative constructions are not covered. Ellipsis are only possible at the utterance level (noun phrases or prepositional phrases).

For parsing a left-corner parser has been adopted, implemented in Prolog. The syntactic structure produced is a tree which guides building the semantic representations. Building semantic representations is done via proper lambda-abstraction and β-conversion [BMM+94, Mus96]. The semantics fragment is not completely isomorphic with the syntactic fragment. Phenomena like modal verbs are left-out in the semantics.

3.1.2 Semantic Construction and Inference

The semantic representations constructed are underspecified discourse representations [BB98]. These are resolved (pronouns, presuppositions, quantifier and operator scope) producing Discourse Representation Structures (DRSs) of Kamp's Discourse Representation Theory [KR93]. The DRSs representing the dialogue's information state are translated into first-order predicate logic, along the lines of [BBKdN99]. Reasoning on these first-order representation is done using the MathWeb distributed inference engines services [FK99], including the theorem provers Bliksem, FDPLL, and Spass. Reasoning tasks cover consistency and entailment checks used for ambiguity resolution [BB98], and dialogue move selection. (The latter consists of checking whether questions on the system's agenda are relevant or not.)

The framework of Discourse Representation Theory (DRT) offers direct means to model discourse and context, and Midas exploit this feature by coding the information state of the dialogue as a DRS. On top of this, Van der Sandt's algorithm for anaphora and presupposition resolution is used to resolve pronouns, proper names, definite descriptions, and other presupposition triggers [VdS92]. The DRS contains also dialogue-act information as proposed by Poesio & Traum [PT98], and therefore covers the task of keeping track of a dialogue history as well. For the user and the system two distinguished predicates are reserved. (So Midas assumes that there is exactly one user.)

3.1.3 Generation

In Midas, an independent module for generation is used, working on the basis of DRS input and string output. This generation module uses the same lexicon as in the user utterance analysis. Besides, there are some fixed expressions that are hardwired with specific dialogue moves (such as basic answers to yes-no questions, greetings, apologizes).

The grammar for the generator focuses on questions. Most rules are duplicates of the analysis grammar, but with direct encoding of the λ-DRT-semantics, based on unification of its λ-arguments. (The analysis grammar assumes underspecified representations, and uses proper β-conversion rather than unification.) The rules for the generator are a subset of those of the analysis module, so Midas understands constructions which it can't generate.

A notorious problem for generators is dealing with the fact that the mapping from the underlying representation to the output string is a one-to-many kind. In Midas' generator this is the case for noun phrases. Alternative readings are suppressed by disallowing the use of pronouns (with the exception of first and second person pronouns), and producing (in)definite description that contain a non-modified noun only. (Modification of nouns is not possible, at present.)

The generator is implemented in Prolog. It has three input parameters: a dialogue act, an utterance DRS, and a context DRS. The output parameter is a list of words or a string. On the basis of the dialogue act the generator decides the basic utterance (e.g., whether it will be a check-question, a yes-no-question or wh-question). It incrementally traverses trough the DRS, and deletes information from it while applying the grammar and lexical rules. Generation of a list of words succeeds when after applying the grammar rules the utterance DRS arrives at an empty set of discourse referents and conditions.

The generator uses the context of the dialogue. Given the DRS of the total information state of a point in the dialogue, the utterance DRS is the current discourse unit under discussion, and the context DRS is the remaining DRS. The context DRS serves to generate context-sensitive expressions such as proper names or definite descriptions, which appear as free variables in the utterance DRSs, but are bound by discourse referents in the context DRS.

3.1.4 Attentional State, Intentional Structure, Dialogue History

The topic of conversation is modelled by implementing "discourse units under discussion", as proposed in the model of Poesio & Traum. This is a stack of discourse referents for DRSs, which abbreviate the basic utterances (made by the user and the system). Focussing

within the sentence level is not implemented. Midas makes no use of linguistic markers that signal topic shifts, neither does it do sub-task identification nor calculate topic prediction.

The task handled by the system is providing a route, in accordance to the application domain (route planning). In order to get the information necessary for calculating this route, the system asks the user for a destination, a departure, and a travel time. These intentions are represented in a *plan-DRS*, loaded into the main-DRS as soon as it is clear that there is a user that wants a route (the latter proposition can be seen as a presupposition for loading this plan). The plan-DRS consists of a series of DRS that paraphrase either questions or actions and intentions to ask or perform these.

Midas does not maintain a record of the surface form of the user utterances, although this is proposed by the Poesio & Traum model to incorporate in the DRS. It does record the underspecified semantic representation of the current question, and by the nature of DRSs, also the entire dialogue structure (in an incremental way). The DRS coding the information state also records the topics that have been addressed (as discourse units). No records of the user's performance are recorded.

3.2 TrindiKit Issues

The information state is specified using only part of the data-types provided by TrindiKit. Midas' natural language processing technology is centered around Discourse Representation Theory, and hence an additional definition of Discourse Representation Structures (DRSs) as TrindiKit-datatype was required, as well as defining (the non-standard) operations on DRSs.

One of the deepest problems encountered here is dealing with ambiguities. The DRSs, as part of the information state, represent the meaning of the dialogue. Regularly, utterances of dialogue participant introduce multiple interpretations. So the actual dialogue is modelled by a set of DRSs. Operations on a set of objects are difficult to handle: what if pre-conditions for applying certain effects to the information state only hold for a subset of the DRSs that are encoded? This issue relates to the discussion on choices of update rules in the SRI system (see Section 4.2). Use of underspecification of representation might play a role here [Poe00].

There are about 40 update rules used in the current version of Midas. There is no difference made in update rules and selection rules, all the rules cover both updates and selections of moves. Already 40 rules make it notoriously difficult to debug the system. To keep the rules under control, they are annotated with a functional type that, when applied, moves from one state to another state. Each state is connected to a set of update rules. Dividing

the set of rules into different states increases tractability of the system enormously.

The TrindiKit control algorithm calls the different modules of the dialogue system in sequence. There is a way to put more control in the update-rules, by declaring some modules as 'resources' to information state updates. This has been carried out in Midas for two reasons: some of the interface variables were of another type Midas was presupposing; and some of the operations of modules directly affect the information state, and hence could not be separated from the update rules (as only update rules are allowed to change the content of the information state).

The first reason is not a serious obstacle for denying the architecture suggested by TrindiKit, it is merely supported by practical circumstances. In fact, some of the ideas currently under discussion in the Trindi-consortium to improve the architecture address the issue to decorate the interface-variables with a flexible type, or even to regard them as modules themselves (where these module take over the book-keeping of the data-types), not just as registers in the overall system.

The second reason for not using one of the interface-variables is of theoretical nature. One of the central ideas underlying Poesio & Traum's theory of information state, upon which Midas' internal representations are based, is to integrate dialogue-acts into semantic representation. Arguably, this makes the use of the interface-variables and 'latest-moves' and 'next-moves' superfluous.

3.3 System Evaluation

In the following the Trindi Tick-list criteria are applied to the current version of Midas (Februar 2000).

Qn 1 *Is utterance interpretation sensitive to dialogue context?*

—Yes. Answers or assertions addressing a different topic than currently posed can resolve questions to be addressed later. Example:

(3.1) S: Where would you like to go?

(3.2) U: I would like to go from Cambridge to London

(3.3) S: Where would you like to start?

The reply (3.2) to the question (3.1) also answers the question (3.3). Midas has a set of distinguished questions marked as "intend-to-ask". A pre-condition for actually posing one of these queries is that it contains informative information. The latter is determined with first-order inference.

Qn 2 *Is utterance interpretation sensitive to deictic context?*

—No. Deictic expressions, such as 'tomorrow' or 'next week' are not dealt with in Midas, because its semantic interpretation module has no access to system time.

Qn 3 *Can the system deal with answers to questions that give more information than was requested?*

—Yes. See the example at **Qn 1a** for illustration. In Midas, answers are valid responses to questions if they are more informative. If more information is given than was requested, the answer is still informative.

Qn 4 *Can the system deal with answers to questions that give different information than was actually requested?*

—Yes. Midas will treat the new information as an assertion, recognizing that the answer was inappropriate, and re-ask the previous question.

Qn 5 *Can the system deal with answers to questions that give less information than was actually requested?*

—No. At present, the interpretation of answers in Midas is carried out at the semantic level. Consider the question:

(3.4) S: Where would you like to go?

Now, answers like

(3.5) U: I want to go to a nice place

(3.6) U: I would like to go somewhere on monday

are recognized as appropriate answers, as they are informative. The appropriateness is highly dependent on the domain of interest. Even in the route-planning domain there are several possibilities: cities, bus-stops, airports, and so on. A pragmatic component, that selects the correct domain in a given scenario, should take care of this task.

Qn 6 *Can the system deal with ambiguous designators?*

—No. That is, partly. Semantic analysis will generate multiple analyses when input is ambiguous. It will choose the most likely reading (based on scores assigned by the various resolution components), otherwise it makes a guess. An extension of the grounding strategy in Midas could deal with this case, by keeping several analyses in store, and pose check-question to the user to find the correct interpretation. Here is an example that illustrates this:

(3.7) S: Where do you want to go?

(3.8) U: I want to go to London.

(3.9) S: Do you want to go to London-Stansted?

(3.10) U: No.

(3.11) S: Do you want to go to London-Heathrow?

(3.12) U: Yes.

Qn 7 *Can the system deal with negatively specified information?*

—Yes.

Qn 8 *Can the system deal with no answer to a question at all?*

—No. If Midas ask something, it will just waits until you hit the return key. It you hit the return button without any input, it will repeat the question currently under discussion.

Qn 9 *Can the system deal with noisy input?*

—Perhaps. Simulating the input of noise to the system can be done by typing rubbish. If there is no syntactic analysis, Midas will re-ask the question.

Qn 10 *Can the system deal with 'help' sub-dialogues initiated by the user?*

—No. Help-questions initiated by the user can only addressed if domain knowledge is present. In Midas, technically it is possible for the user to ask questions. These are answered with respect to the knowledge so far recorded in the dialogue. This holds also for the 'non-help' sub-dialogues.

Qn 11 *Can the system deal with 'non-help' sub-dialogues initiated by the user?*

—Yes. Questions initiated by the user are answered by Midas with respect to background knowledge and data so far recorded in the dialogue.

Qn 12 *Does the system only ask appropriate follow-up questions?*

—No. At present, Midas has a set of questions that it intends to ask. A precondition for asking question is that the information it contains, should be informative with respect to the current dialogue. There are no internal relations specified between these questions. To fulfill this criterion, Midas has to be enriched with plan-DRSs that relate questions to several domains and to other questions. This is currently under development.

Qn 13 *Can the system deal with inconsistent information?*

—Yes. The semantic component recognizes contradictions. Inconsistent interpretations are not considered. If no interpretation is left, Midas will restart the dialogue.

Qn 14 *Can the system deal with belief revision?*

—No. Belief revision is required if inconsistent believes arise. In dialogues, these could be triggered by a correction of one of the participants, or by a misunderstanding of the system. Consider the following dialogue:

(3.13) S: Where do you want to go?

(3.14) U: Eh, I want to go to Paris.

(3.15) S: When do you want to go?

(3.16) U: No, I don't want to go to Paris. I want to *start* in Paris.

Midas will detect the inconsistency, and as a consequence, start a whole new dialogue, thereby forgetting the information that the user wants to start in Paris.

Qn 15 *Can the system deal with barge-in input?*

—No. The present system architecture doesn't allow this, and is based on 'turns'.

Qn 16 *Is it possible to get a system tutorial concerning the kind of information the system can provide and what the constraints on input are?*

—No. There are no such facilities.

Qn 17 *Does the system check its understanding of the user's utterances?*

—Yes. The system works in three different modes concerning grounding of utterances. Optimistic grounding: utterances of the user are not checked on the system's understanding. Cautious grounding: utterances are only checked on the system's understanding if they got a low confidence score by the parser. Pessimistic grounding: utterances are always checked on the system's understanding,

3.4 Conclusion

It is interesting to interpret the evaluation results of the complete systems in the light of the Trindi DME-architecture. That is, to answer the two questions how much is done with the DME, and how much is system-dependent, i.e., depending on other modules than provided by TrindiKit, or theory-dependent. This is difficult to say. It seems that most of the Tick-list criteria concern DME-external components. This makes it a wrong instrument

for evaluating the DME (and TrindiKit in particular), but not necessarily a bad instrument for evaluating complete dialogue systems.

At least one issue, not under implementation at present, might be greatly affected by the DME, namely dealing with discontinuous input. At the moment the DME works on the basis of processing 'turns'. One participant produces several utterances (together forming the current turn), the other participant analyzes all the utterances, and responds in a next turn, then the other participant analyzes the new turn, and so on. An asynchronous architecture, where different modules can access and change different parts of the information state, seems to be a natural candidate.

Chapter 4

SRI Autoroute Dialogue Demonstrator

In this chapter we provide some evaluation of the SRI Autoroute Dialogue Demonstrator and the TrindiKit model in which it is implemented. First, we provide a general description of the demonstrator system, in particular focussing on parts other than 'Conversational Games Theory' (CGT) which was described more fully in D2.1. Secondly, we discuss some important conceptual and implementational issues that arise directly with respect to implementation in the TrindiKit. That is, the evaluation we are here interested in is primarily the value of the Trindi dialogue model perspective and the TrindiKit implementation. Finally, we re-visit the 'tick-list' of evaluation points (as used in the Trindi project's initial survey report D1.3) and re-apply it to our implemented system. The exercise is valuable not just in throwing further light on the abilities and limitations of the current demonstrator but also in raising important questions about how a CGT-based dialogue manager ought to be further developed.

4.1 System Description

The TrindiKit implementation of CGT for Autoroute can be described under the following headings: Parsing and Semantic Analysis, Dialogue Management and Generation. Dialogue Management in the SRI demonstrator has been extensively discussed in document D2.1 and so it is given a rather more cursory treatment here. In the following sections, we particularly highlight the role of inference and dialogue context where it influences the operation of other system components.

4.1.1 Parsing & Semantic Analysis

The parser for the CGT Autoroute demonstrator is a simple phrase-spotter. It employs a top-down left-to-right algorithm in a search for a string of substrings which covers the total input string. Each substring can either be a single token (which is ignored semantically) or is a substring which contributes one or more propositions to the set of propositions that constitutes the meaning of the input string. For example, the string *I want the quickest route to get me to cambridge by four p m* is divided into substrings as follows

(I) (want) (the) (quickest) (route) (to) (get) (me) (to cambridge by four p m).

Of these substrings, *quickest* generates the proposition `tripmode(quick)`, and *to cambridge by four p m* generates two propositions `endtime(16:0)` and `to(cambridge)`. All the other substrings – *I*, *want* etc. – are semantically ignored. They contribute no information to the propositional value of the input.

The algorithm is left-to-right in that it scans the input string from left-to-right looking for an initial substring which generates a propositional contribution. If one is found, the algorithm simply recurses on the remainder of the input string. If none is found, the first word is discarded as semantically valueless and then the algorithm recurses on the remainder of the input. The algorithm is top-down in that, when it looks for an initial substring, it looks for more informative substrings first. For instance, one category of substring is simply *clock-time* and in our example *four p m* is a clock-time. Another category is *place* and *cambridge* is a place. Each of these substrings can contribute propositional information. In the absence of other information, places are assumed to be destinations and times are assumed to be departure times. However, the covering substring *to cambridge by four p m* does provide more information than the two smaller substrings because *to* and *by* indicate that the *destination* is cambridge and that the *arrival time* is four p m. The amount of "default" information required is smaller. Consequently, the algorithm searches for the more informative substrings first.

The algorithm is also dialogue-sensitive in that it knows the current propositions under discussion when it parses the next user input. This is to enable the parser to interpret fragmentary input or elliptical input in the current dialogue context *when a default inference is required*. If the only available initial substring of the input string underdetermines a proposition, as in a simple one-word utterance such as *cambridge*, then the dialogue context can be used to fill in the proposition.

The following examples illustrate the parser's operation.

InputString	Context	Meaning
cambridge	null	`to(cambridge)`
cambridge	lb(x,from(x))	`from(cambridge)`
cambridge to london	null	`from(cambridge),to(london)`
cambridge to london	lb(x,from(x))	`from(cambridge),to(london)`

In the first two examples, the substring is insufficient to determine a full propositional meaning and so, in the null context, *cambridge* is interpreted as a destination. If the context includes the issue of journey start points, however, then `from(cambridge)` is preferred. In the second two examples, the substring is sufficient to determine itself the propositional meaning and so in both the null context and the non-null context, the meaning is `from(cambridge),to(london)`.

The parser has limitations, of course, notably in the following respects. First, although it possesses some ability to ignore input that it cannot make use of, the input that it can make use of must comprise a substring of the original input string. For example, the input string *cambridge thanks to london* is not itself a meaningful substring. Consequently, the parser considers the two substrings *cambridge* and *to london* but it considers them in isolation from each other. Although *thanks* is discarded semantically, it still imposes an interpretation barrier within the string. The parser does *not*, for instance, act just as if the string were *cambridge to london*. Strings are broken up into substrings but no inference is made concerning the relations between substrings. Of course, it appears to be a substantive issue what interesting relations there might be to discover. If a string mentions a destination and a time but there is no string covering both these concepts, then in what circumstances is it reasonable to infer that the time is an arrival time? Does it depend on the amount of uninterpretable intervening material or whether we know the system is operating with imperfect speech recognition? What influence should the dialogue context have in this circumstance? Although these are difficult questions, one relation that might be very useful to discover is *mutual exclusion*.

The current parser attempts no such reasoning at all and consequently, if the dialogue context concerns the issue of destinations, then *cambridge* will be interpreted as a destination (using dialogue context) and so will *to london* (using the linguistic knowledge derived from *to*). The parser cannot reason about the consistency of the interpretations it generates. The parser cannot determine that `to(cambridge),to(london)` is inconsistent. Again, of course, it is a substantive issue what one should do even if the inconsistency were discovered. A method of weighing contributions would be required. In our current example, one would need to be able to reason that *cambridge* was interpreted as a destination because of the current dialogue context, whereas *london* was interpreted as a destination because of explicit linguistic marking via *to*. In these circumstances, it appears the linguistically marked case should take preference. However, the combination of all these factors makes interpretation increasingly complex.

possible rule Interpret *cambridge* as a starting place (destination) if it is discoverably linguistically marked so, or if the dialogue context concerns starting places (destinations); otherwise interpret it as a destination unless this generates an inconsistency in which case interpret it as a starting place

Furthermore, such a rule only covers one sort of immediately apparent inconsistency where the evidential basis (dialogue context versus linguistic cues) differs and favours one interpretation.

4.1.2 Dialogue Management

The Dialogue Management module of SRI Autoroute demonstrator has been described elsewhere (D2.1). Here, we will confine ourselves to a very brief review.

The dialogue manager consists of two parts: a rational agent and a conversational games player. The rational agent possess a limited amount of knowledge about the world and this includes knowledge of what propositions have been agreed in the dialogue so far. The agent also possesses knowledge about how to change the world. The means of changing propositions agreed so far is to play conversational games. The agent has a goal (show the user a route satisfying his parameters) and he reasons his way from the goal, his knowledge of the way the world is and his knowledge of how to change it to a plan of action. The plan will include playing certain conversational games. The agent plans to agree propositions concerning the journey's origin, destination, time, and mode (i.e. whether the quickest or shortest route is required). Then he plans to consult a database of routes before informing the user of the best route.

Conversational Game knowledge is encoded as simple grammars in the form of transition networks. The networks define the valid moves that can be made at any point in the game. Moves and games are categories that contain both a general type (e.g. question, reply, acknowledgment) and a content (e.g. 'whether the destination is london', 'that the start time is 4 pm'). The Conversational Game player determines whose turn it is to say something, and if it is the user's, it obtains input from the user, analyzes which possible dialogue moves it can constitute and chooses one according to a utility-based preference mechanism. The turn then passes to the system. On the system's turn, a list of possible next moves is generated and again selected from via the same preference mechanism. The move is then realized as an output utterance.

Although we do not discuss the issue further here, the game player maintains at all times the set of possible game parses. If the currently most preferred parse cannot be extended to cope with the new input (e.g. the user says something unexpected) then that parse can

be discarded and another possible analysis of the dialogue so far is chosen.

The input to the dialogue manager is a representation of the meaning of a user utterance, for example, `to(cambridge)`. The first job of the dialogue manager is to compute the *dialogue move(s)* that it represents. Some moves, e.g. acknowledgments, are decipherable purely on account of their content. Others depend on a relation to the current propositions under discussion. For example, an utterance meaning of `to(cambridge)` counts as a *reply* only if the propositions under discussion take a certain form. Replies must answer questions. Consequently, if the previous question was *Where are you going from* and the user utters *to Cambridge* then this will not be analysed as a reply. However, there are other possible analyses. It may be that the user is simply giving the system new information (and ignoring the system question). In principle arbitrary inference can be used to determine a dialogue move. In fact, only very small and highly specialized inferences are used. For example, one inference relates the meaning of two propositions sharing the same function (e.g. *to(cambridge)* and *to(london)*). In this way, if *to(london)* is under discussion (being confirmed) and the next utterance is *to(cambridge)*, then the latter can be interpreted as a 'reply-modifier' move.

4.1.3 Generation

The language generator in the demonstrator program is a simple template filling approach. For example, if a proposition of the form *from(X)* is to be confirmed, then an utterance of the form 'You are starting from X. Is that correct?' is generated.

4.2 TrindiKit Issues

The Conversational Games Theory model of dialogue invoked in the SRI Autoroute demonstrator is a two-level theory of dialogue. The theory invokes general *rational agency* notions to explain macro-structure in dialogue but the idea of shared knowledge of conversational game structure to explain micro-structure. It therefore represents a point in between the two pictures mentioned in Chapter 2. In the classical plan-based picture individual utterances are planned. In CGT, only conversational games (conventional types of sequences of utterances) are planned. Within a game, utterances are subject to a state-transition model. In the finite state dialogue models, the whole dialogue is subject to such a model.

The two-level theory is easily implemented inside the Trindi Model. The information state is essentially divided into two parts and there are two sets of update rules, each set updating one part of the information state. Separation is ensured by specifying `emptyRec(AGENDA)`

as a precondition in each of the 'rational agency' rules, *i.e.* if a game is currently underway, then you cannot deliberate about other games to play or change one's global plan. This is sufficient to implement CGT as currently conceived. One potential advantage of the Trindi Framework however is that it would not be a significant implementational task to undo the separation. One could experiment with interactions between "within-game decision-making" and "between-game decision-making". Certainly, in the original implementation of CGT supplied at the beginning of the project, such a re-working would have been difficult without large amounts of re-implementational work elsewhere. The 'flat' nature of decision-making in the Trindi model with a single set of update rules all operating on a single notion of information state here provides some flexibility for model development. The possible disadvantages include the fact that the different functionalities may be not be readily decipherable by, for example, system maintainers who wish to understand the precise functions of various parts of the flat structure. Also, from a strictly implementational viewpoint, the update algorithm which operates upon the single set of update rules may not be optimal for all the different functions which it is used to compute. That is, it might very well be more efficient to have specialized representations and algorithms for specialized pieces of work. However, in a research and educational tool of the nature of the TrindiKit, this point has less significance than if it were directed at commercial systems development.

The general Trindi model provides also for a set of rules to update an information state given an input dialogue move (or moves). The model is however agnostic concerning the precise point at which a choice is made concerning which move was made and whether and how choices can be un-made or re-made. In one natural instantiation of the model, the choice of which move was made by an utterance is taken prior to operation of the update rules. The update rules then encode how to update the information state given a particular choice of move. That is, one first recognizes an utterance as constituting, for example, a question and then applies the update rule for questions (or, one of the rules for questions, if the rules are also context sensitive) thereby updating the information state. In this natural instantiation, there is a certain degree of early commitment implied in the early choice. For how can the system undo or redo an update, if a problem arises later which is caused by an earlier incorrect choice? In order to allow this, the information state must carry around within it not just the new information generated by the early choice but also information that might be required if the choice is to be un-made or re-made.

The point is particularly salient for the SRI CGT dialogue manager. CGT posits conversational games and conversational moves. Games are constructed out of moves and, indeed, it is knowledge of this structure that is shared between dialogue participants. In SRI's development, these structures are formulated as fairly simple transition networks and the discovery of the structure becomes a parsing problem. Choice of parse is not however something that is committed to early. The best analysis of one dialogue move depends in part on what the *next* dialogue move is analysed as. Some sort of backtracking is therefore required. In order to accommodate this picture, the update rules in the CGT Trindi model

encode the parsing process over conversational games. That is, the information states that are maintained over a dialogue include the state of the current and possible states of alternative parses and the update rules encode the search algorithm over that space. The preconditions of the update rules determine how the search will be expanded and the effects determine the results of the expansion. Each of the possible new parse states will also encode an update effect resulting from an analysis of the latest dialogue move. For example, there may be one parse which determines that the propositions under discussion is a certain set X and another which determines that the set is actually Y. But the updates which generate X and Y are not themselves encoded in Trindi update rules. They are encoded elsewhere in a resource. Of course, in order to progress the dialogue one of the options must indeed be chosen. Consequently, there is a utility-based preference routine which subsequently sorts the set of choices generated by the update rules. Future action is based on the top choice, but the alternatives are not thrown away. They will be further developed along with the current choice in the light of new data. Later, one may find it has become top choice after all. In fact, making the choice (i.e. sorting the set of possibilities) is encoded in the update algorithm. It could equally be the subject of a further update rule encoded in such a way that it only operates after the other updates rules have been applied.

The general point to be made from the discussion is the substantive issue of the distribution of labour amongst the Trindi components. In the very last case discussed – i.e. should sorting the parsing possibilities be encoded as the last update rule to apply or as the last part of the update algorithm? – nothing very much appears to depend on the answer. The earlier case – should the update effects of individual dialogue acts be encoded in the update rules as in the "natural instantiation" ? – the difference appears to be more significant. We wish to avoid the idea of "early committment" to a particular analysis. Of course, it may still be possible to encode the update effects directly in update rules. They might be encoded in another *sort* of update rule. In this case, our earlier remark on the flatness of the update rule formalism possibly masking quite different functionalities would again be relevant.

4.3 System Evaluation

In this section we revisit the 'ticklist' of evaluation points (see Section 2.4) with regard to the current CGT demonstrator system.

Qn 1 *Is utterance interpretation sensitive to dialogue context?*

—Yes. In the demonstrator system, there are two different types of interpretation. First, a parser extracts a semantic representation of the content of the utterance. Secondly, the conversational game player classifies the utterance as a dialogue move. The former process is context sensitive in a simple way, as explained above (section 4.1.1). If a substring is found, e.g. *cambridge* which is insufficient in itself to determine a propositional value, then the context of the current propositions under discussion may be used to aid interpretation. Of course, this sensitivity is only available within conversational games since only then are there propositions under discussion. Dialogue move interpretation is also potentially sensitive to context. For example, utterances which follow questions are only deemed to be *replies* if their content suitably matches that of the question.

In an interesting way, the context is also sensitive to utterance interpretation in the CGT framework. The conversational games player maintains a set of alternative parses of the current dialogue. One of them is most preferred. The most preferred one determines, for example, what the system next utters. However, if a new input cannot be interpreted with respect to the most preferred parse (or even if it can, but the interpretation is not considered likely), then an alternative context may be switched to.

Qn 2 *Is utterance interpretation sensitive to deictic context?*

—No. Deictic expressions are not currently handled by the Autoroute demonstrator.

Qn 3 *Can the system deal with answers to questions that give more information than was requested?*

—Yes. The definition of a reply move in the current CGT implementation is an utterance whose content *at least* answers the question. Therefore, if the propositions under discussion after a question has been raised are `lb(x,to(x)` and the next utterance content is `to(cambridge),from(london)`, then this is deemed a reply whose move content is `to(cambridge),from(london)`.

Qn 4 *Can the system deal with answers to questions that give different information than was actually requested?*

—Yes. The conversational games player can deal with such cases by treating them as instances of an `interruption` game. An interruption game is defined as one player making a move deemed to be of no importance and the other player starting an information-giving game.

If the system issues the question *Where do you want to go?* and the user replies *I'm leaving Cambridge at four*, then the system finds that its hypothesis that a question game was under way has become untenable. The alternative analysis, that an interruption game is under way becomes preferred. On this analysis, the system's own utterance is analysed as being of no importance and the user's utterance is deemed as beginning an information game. The system will choose to complete this game by issuing an acknowledgement.

(In point of fact, there are several alternative hypotheses for the system to consider. One possibility is that the user's utterance should be deemed unimportant and that the system itself should interrupt. The weights in the utility based preference mechanism are set to disprefer this option strongly. Another option is simply that the user's utterance was not heard correctly. In this case a 'pardon' game is under way and the system should utter 'pardon'. Again this is dispreferred because meaningful content was extracted from the utterance).

Qn 5 *Can the system deal with answers to questions that give less information than was actually requested?*

—No. The CGT-based dialogue manager has no explicit means to deal with such cases. It is a matter of some interest how such cases should be dealt with. For instance, do such utterances count as *replies*, or perhaps *replies of a special sort*? In this cases, one would need to discover or devize a sequence of follow-up dialogue moves for completing the game. Theoretically, such an account is probably not tenable since dialogue moves are intended to be clearly publically recognizable as such and the structure of the games shared knowledge. If a hearer categorized an utterance as a special sort of less-informative reply, is it clear that the speaker does so too - either when he issues it or after hearing the next dialogue contribution?

An alternative is to analyse the less informative utterance as a simple information giving move which is not a continuation of the current question game. In this case, the move could be acknowledged and the current game completed. Then the rational agent would have to compute what its next best course of action should be: a relevant follow-up question. However, no such facility is currently implemented.

Qn 6 *Can the system deal with ambiguous designators?*

—No. The CGT-based dialogue manager currently has no means to deal with such cases. The parser delivers its best guess at a propositional value to the dialogue manager. The best guess cannot currently be underspecified with regard to semantic content.

Qn 7 *Can the system deal with negatively specified information*

—Sometimes. The CGT-based dialogue manager uses negative information at only one point—in the confirmation routines within a question game. The parser itself merely notes that an utterance contains a negation. Negation is therefore treated as a discourse marker (in the same way as an acknowledgment) rather than as having semantic value with scope over content.

Qn 8 *Can the system deal with no answer to a question at all?*

—No. The system currently always expects the user to answer questions.

Qn 9 *Can the system deal with noisy input?*

—Yes. The parser, albeit a fairly simple one, functions as a form of phrase-spotter. Consequently, it is not crippled in the face of noisy input. Naturally, its degree of success depends to a large extent on the degree of noise.

Qn 10 *Can the system deal with 'help' sub-dialogues initiated by the user?*

—No. The system has no such facility.

Qn 11 *Can the system deal with 'non-help' sub-dialogues initiated by the user?*

—Yes. The system can deal with such dialogues using the model of "interruption games" explained above.

Qn 12 *Does the system only ask appropriate follow-up questions?*

—Yes. The system architecture is designed in order to facilitate dialogues which do only ask appropriate follow-up questions.

Qn 13 *Can the system deal with inconsistent information?*

—No. The system does not perform any such reasoning.

Qn 14 *Can the system deal with belief revision?*

—Yes. The theoretical motivation behind the Autoroute demonstrator system posits two different levels of "belief representation". Within a conversational game, there is a set of propositions under discussion and these may be revised but only according to the moves permitted by the rules of the game. The game can be considered as a rule-governed method for revising the set of propositions under discussion. The point of the game is establish certain propositions as agreed. Once a proposition has been agreed however, subsequent revisions may still occur; but this will be at the level of game-playing itself and theory simply posits a "rational agent" who will update agreed commitments in a rational way. The current implementation only supports a simple "overwrite" revision at this level. No inconsistency is detected. In the following example, the system and the user agree that the destination is London but the user then immediately overturns the agreement. The system does not detect any problem and simply updates its belief about the destination with the new information. (That this is so is demonstrated by the final route provided).

```
 1.Where do you want to go to?
 2.london
 3.You are going to london. Is that correct?
 4.yes
 5.Okay.
 6.Where do you want to go from?
 7.I want to from cambridge to oxford
 8.You are starting from cambridge. Is that correct?
 9.yes
10.You are going to oxford. Is that correct?
11.yes
12.Okay.
13.When do you want to travel?
14.I want the quickest journey at five
15.You want the quickest journey. Is that correct?
16.yes
17.You will start your journey at 5:0. Is that correct?
18.yes
19.Okay.


Time Distance Route
```

```
00:07   3.1  A1307 -> A428
00:07   3.6  Cambridge-West 14 A428 -> M11
00:33  46.5  Shell S05-8922 M11
00:46  69.2  M25-M11 M11 -> M25
00:50  74.9  Shell S05-1392 (5.1 km) B1393 M25
00:50  75.0  Shell S05-8820 (4.7 km) M25
00:02  94.9  M25-A1(M) M25
00:24 131.0  M25-M40 M25 -> M40
00:51 175.6  M40-A40 M40 -> A40
00:58 184.7  A40 -> A420
00:01 185.8  Shell S05-9526 A420 A420
00:09 189.2  Oxford (GB)
```

(The above example illustrates the Autoroute system hooked up to a web-based route planning service `www.shell.com/euroshell/routeplanner` provided by Shell UK)

Qn 15 *Can the system deal with barge-in input?*

—No. The user is not permitted to barge-in over system utterances.

Qn 16 *Is it possible to get a system tutorial concerning the kind of information the system can provide and what the constraints on input are?*

—No. The system provides no such tutorial information

Qn 17 *Does the system check its understanding of the user's utterances?*

—Yes. By default, the system does perform check its understanding of the user's utterances by asking a yes-no question concerning the content which it understood. If a user utterance contains more than one proposition, then multiple checks may be required.

```
S.Where do you want to go from?
U.I want to from cambridge to oxford
S.You are starting from cambridge. Is that correct?
U.yes
S.You are going to oxford. Is that correct?
U.yes
S.Okay.
```

4.4 Conclusions

Current demonstrator development has had two primary motivations: formal and implementational development of CGT itself; and "proof of concept" of the TrindiKit dialogue model and development environment. Both of these motivations have been been largely satisfied by the current development. Although, we currently have no future plans for developing the CGT demonstrator further. There are of course many ways in which development could be usefully pursued.

First, whilst the formal development of CGT may proceed usefully along the lines pursued here, it becomes an increasingly urgent matter to ensure that the formal development is empirically verifiable and verified.

The Tick-list evaluation points and the general TrindiKit issues raised earlier also raise some important issues for future development.

First, how straightforward is it to define and identify dialogue moves and games in increasingly conversational environments? The simplest human-computer dialogues are ones where computers supply questions and humans either answer them in a fully satisfactory way or they fail to do anything at all. Moves are questions or answers and games are sequences of such. More flexible behaviour requires more interesting moves and games. In the CGT development, either the set of moves and games itself can be extended or the definitions of particular moves and games can be extended. In order to allow "more information in answers than was requested" (ticklist point 2), the second strategy was followed. A 'reply' is deemed to be anything that *at least* answers the question. For "different information than requested" (ticklist point 3), the first strategy is adopted. For "less information than requested", no strategy is adopted although the second, as discussed earlier, appears more plausible. The issue of "identifying the possible functions of utterances" is of course one that arises for any dialogue theory. With regard to the current version of CGT, there is in addition the issue of "identifying the single function of an utterance" (the function may be arbitrarily complex but CGT only permits an utterance to be classified one-way) and the issue of "discovering the shared known structure over moves". As move-classifications become increasingly complex, it may be that one is less willing to identify clear known and shared game-structures over those moves. Such a development would not invalidate CGT itself but it would mean that most of the more flexible behaviour would have to be accounted for by the rational agency portion of the CGT model and that games themselves would account for less and less.

The current instantiation of the theory also imposes some limits on the degree of flexibility allowed, notably, that only one function per utterance is currently permitted (as noted above), that the semantic input to the dialogue manager should be fully determinate (ticklist point 5) and that a turn-taking model is employed (ticklist point 7). The first

point is more of a conceptual one. In part, it derives from a picture of dialogue in which participants continually try to identify what the *point* of other's contributions are. CGT is partly an attempt to move away from this picture. In annotating a game of chess, for example, one can of course try to decide at each point the functions of each move. It is also obvious that each move may have several functions. The move may threaten one piece, defend another and suggest a whole line of attack. First, however one must identify the move ("pawn to e4"). CGT is an attempt to identify moves at a higher level than simply identifying the surface locution ("he said the following words 'I want a route to Cambridge") or its meaning.

The second and third limits (determinate semantic input and turn-taking) are perhaps the more significant ones. *Should* semantic input be determinate? Would it be sufficient for the parser to deliver a set of possible interpretations, rather than a single best guess, to the dialogue manager. The answer will clearly be 'no determinism' if there are cases where the parser/semantic interpreter cannot make the right choice early. Unfortunately, it is just not clear whether parsing and semantic interpretation are in principle restricted in the information sources they can draw upon. In the general Trindi model, they can invoke any information available also to later components. In the current CGT demonstrator, the parser and semantic interpreter, as described earlier, has access to the 'latest question' asked but this is a somewhat arbitrary restriction. The more important consideration appears to be that the decision should not be taken 'early' by *any* module. The best response to an utterance which is unclear in some aspect is to clarify it - that is, the dialogue manager should respond to the very unclarity. Consequently, merely passing on a set of interpretations to a dialogue manager will not be sufficient unless the dialogue manager can respond to the set as a whole and does not evaluate each member on its own. The CGT dialogue manager, for example, already maintains a set of possible dialogue parses. It would not be difficult to extend that set to include different resolutions of ambiguous inputs. It would however be a significant effort to make it's next action depend, not on what it judges the best member of that set, but on relations between members of the set.

The third limit concerning strict turn-taking poses interesting questions for the current demonstrator system. It ought to be possible to operate the parsing, generation and dialogue interpretation modules in more of a "blackboard mode" of operation than they do currently. Parsing could operate whenever and as soon as typed or spoken input was added to the blackboard. Dialogue interpretation could operate whenever parsing posted its results to the blackboard but it would also need to be able to restart (or at least readjust) if it had not completed its operations and more input from the user was posted. The resulting picture is a rather minimal one - the system is simply yielding the floor whenever asked to. There are obvious dangers in such an approach – if the system always yields the floor but always takes too long in its own deliberations, the user might never receive any feedback and believe nothing is happening at all. Consequently, one might wish the system to insist on holding the floor sometimes.

The Trindi model and TrindiKit environment undoubtedly provide a unifying structure within which such questions can be usefully pursued. It should be clear from the discussion above that the general environment does not answer those questions. Even merely separating out a 'semantic interpretation module' from a 'dialogue move interpretation module' can become a contentious issue. Nevertheless, a more abstract level of description of dialogue systems in terms of their information states and information flows is clearly a move in the right direction for more readily understable and comparable systems.

Chapter 5

EDIS

This is a description of version 1 EDIS, which implements a number of key aspects of the Poesio-Traum Theory (PTT) of Information State (IS) and update rules, using the TrindiKit. A description of the main points of version 2 can be found in [MTP00].

For background information on the update rules, see [CLM+99]. The main contents (section 5.1) describe the model of IS, the update rules, and the associated update algorithm. It was necessary to extend the DME in a number of (typically small) ways. Section 5.2 outlines these additions.

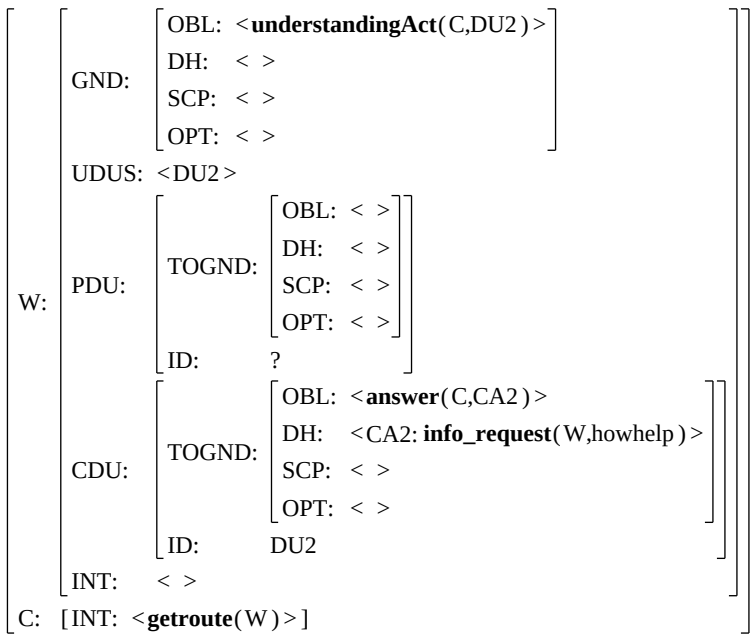
5.1 System Description

The main aspects of PTT which have been implemented in EDIS concern the way discourse obligations are handled and the manner in which dialogue participants (DPs) interact to add information to the common ground. The following subsections look at the model of IS which is assumed, the update rules used, and the update algorithm assumed by the rules.

5.1.1 Information States

The implemented structure of ISs is fairly close to the version in [CLM+99], with a number of simplifying assumptions. The following is an example representing the relevant information following the first utterance (*how can I help*) in the autoroute test dialogue we have been working with :

(5.1)



The IS has two main parts, representing the information associated with the system (Wizard) and that associated with the user (Caller). Normally these sub-structures would be identical; however, because in this case we are not attempting to model misunderstandings arising from the dialogue participants having differing views on what has been grounded, it is only necessary to preserve a separate INT field for C's intentions.

The main part of the IS has information on the grounded material (GND), on the ungrounded information (UDUS), and on W's intentions (INT). EDIS assumes that contributions to the dialogue are stored in the form of DISCOURSE UNITS (DUs); for each new utterance, a new DU is created and added to the IS, and there is no independent limit to the number of DUs which an IS can contain.

DUs and the GND field contain four elements, representing obligations (OBL), the dialogue history (DH), social commitments (SCP) , and 'options' (OPT). The latter is really a marker for future work and we shall ignore it here. The other attributes contain the immediately relevant information; the obligations, the dialogue history, and a list of propositions to which the participants have become committed during the dialogue.

The implementation differs from the full theory in that only two DUs are retained; the current DU (CDU) and the previous DU (PDU). We keep the previous information around when processing new input in order to retain the basic assumption that UNDERSTANDING ACTS (acknowledgments, and suchlike) must be performed before the information in a DU becomes grounded. In making the change from an unlimited number of DUs to just two, we could also do without the explicit representation of the ungrounded information (UDUS); however, in the analysis of more complicated dialogues it is necessary to preserve

this information in order to return to previous DUs, so we maintain the field as a marker for future work.

Looking again at (5.1), the values of the OBL and SCP attributes are lists of 'dialogue action types', while elements in the dialogue history are 'dialogue actions'. A dialogue action type has three attributes: a PREDICATE, a DIALOGUE PARTICIPANT, and a list of ARGUMENTS. In GND.OBL in (5.1), the predicate is **understandingAct**, the participant is C, and there is just one argument, DU1, which identifies the information in CDU by referring to its ID. In (5.1) the common ground contains the information that C has an obligation to **answer** conversational act 1, and this points to the appropriate information request in the DH list by means of the ID number. Note that the dialogue act in the dialogue history contains a representation of the proposition which was the content of the original question as a simple string.

5.1.2 Update Rules

The structure of ISs in the P-T model was outlined in the previous section. Now we are in a position to look at the update mechanisms which serve to map particular ISs into new ones. To make this discussion concrete, we shall use the test dialogue from the SRI Autoroute corpus which is used to introduce the Trindi annotation schemes in [CLM+99]. The dialogue in question has been 'cleaned up' to some extent for use in this context, and we assume it goes like this:

(5.2)

W[1]: How can I help?
C[2]: A route please
W[3]: Where would you like to start?
C[4]: Malvern
W[5]: Starting in Great Malvern?
C[6]: Yes
W[7]: Where do you want to go?
C[8]: Edwinstowe
W[9]: Edwinstowe?
C[10]: Yes
W[11]: Edwinstowe in Nottingham?
C[12]: Yes
W[13]: When do you want to leave?
C[14]: 6 p.m.
W[15]: Leaving at 6 p.m.?
C[16]: Yes
W[17]: Do you want the quickest or the shortest route?
C[18]: Quickest
W[19]: Please wait while your route is calculated

The current system can automatically process this dialogue. To get this to work, we assume that before the dialogue starts W has the intention to ask C what kind of help is required, and also that C has the intention to find a route. The initial IS is therefore as shown below:

(5.3)

$$\begin{bmatrix} \text{W:} & \begin{bmatrix} \text{GND:} & \begin{bmatrix} \text{OBL:} & < > \\ \text{DH:} & < > \\ \text{SCP:} & < > \\ \text{OPT:} & < > \end{bmatrix} \\ \text{UDUS:} & < > \\ \text{PDU:} & \begin{bmatrix} \text{TOGND:} & \begin{bmatrix} \text{OBL:} & < > \\ \text{DH:} & < > \\ \text{SCP:} & < > \\ \text{OPT:} & < > \end{bmatrix} \\ \text{ID:} & ? \end{bmatrix} \\ \text{CDU:} & \begin{bmatrix} \text{TOGND:} & \begin{bmatrix} \text{OBL:} & < > \\ \text{DH:} & < > \\ \text{SCP:} & < > \\ \text{OPT:} & < > \end{bmatrix} \\ \text{ID:} & ? \end{bmatrix} \\ \text{INT:} & <\textbf{info\_request}(\text{W,howhelp})> \end{bmatrix} \\ \text{C:} & [\text{INT: } <\textbf{getroute}(\text{C})>] \end{bmatrix}$$

We also assume that W has the turn, which means that a sequence of actions are performed, beginning with the *selection* process, which employs one of three selection rules and the default algorithm. The `inforeq` selection rule fires in the presence of the first intention and triggers an utterance directly.[1] The actual srule is:

```
srule(select(ask), _,
       [ fstRec(w^int, INT),
         INT:valRec(pred,inforeq)],
       [ popRec(w^int) ],
       [ INT ]
     ).
```

This just checks to see if the first intention is an **inforeq**, and if so, it pops the intention and sets `next_moves` to be the relevant question. The next action associated with the system turn is to *generate* a stored string which expresses the element in `next_moves`. This string is then written to the screen by the *output* module, which also sets `latest_moves` to be whatever was in `next_moves`.

In the current implementation, two more stages have been added to this *select*, *generate*, and *output* cycle to constitute a full 'system turn'. The first is an *acknowledge* step following *output*; this allows an implicit acknowledgment of a previous utterance to be included in the moves associated with an utterance. EDIS assumes that acknowledgments (Understanding Acts) are necessary for an utterance to be grounded, and so, for example, the utterance by W of *Where would you like to start?* in (5.2) contains the necessary implicit acknowledgment of C's utterance [2]. When running the system's turn, then, it is necessary to include the information about which previous utterance is being acknowledged (if any). The final step in the turn runs the update rules to produce the next IS, and so the full macro for 'system turn' is:

```
st(ACT) :-
  select,
  generate,
  output,
  acknowledge(ACT),
  update.
```

If `ACT` is instantiated, `latest_moves` will contain the appropriate acknowledgement as well as a representation of the core acts in the utterance. So, running the system turn produces

[1] The production of utterances in the full PTT model is a good deal more complicated, relying on various kinds of deliberation.

the utterance *How can I help?*, sets `latest_moves`, and maps the IS in (5.3) into (5.1). In this case, as the utterance is the first in the dialogue, there is nothing to acknowledge, and `latest_moves` will just contain the information request:

```
stackset([record([pred=inforeq,
                  dp=w,
                  args=stackset([record([item=howhelp])])])])
```

An account of the update algorithm used in the implementation is provided in section 5.1.3 below. In the deliverable, there are four stages to the updating process which can be summarised as:

(5.4)

1. Create a new DU (CDU) and push it on top of UDUs.
2. If a backwards grounding act is observed, merge CDU and ground, update UDUS, and record the act in ground
3. If any other type of act is observed, record it in the dialogue history in CDU and deal with it as specified by the particular rule which handles the act in question.
4. Apply update rules to all parts of the IS which contain newly added acts.

Note that the final step refers to DRSs; here this means the appropriate parts of the record structure representing the IS. The process of dealing with the specific acts handled by steps 2 and 3 is described below with respect to particular update rules; the main point here is that the four steps above suggest an algorithm for dealing with updates which has been implemented fairly closely in the current system. Clearly, the suggestion is that the steps must be carried out in sequence.[2] Firstly, creating a new DU is interpreted as copying the contents of CDU to PDU and incrementing the ID number in CDU. The second and third stages deal explicitly with the contents of `latest_moves`, as discussed below, and the fourth cycles through the updating process in case recently added acts have further implications. Before looking at the update algorithm, we shall discuss the actual update rules which map (5.3) into (5.1), beginning with the rules which create a new DU as suggested in step 1 in (5.4).

```
urule( makeNewDUandCopyCDU, ruletype1,
       [ valRec(w^cdu,CDU) ],
```

[2]This means that the default DME update algorithm cannot be used as it assumes that the first, and only the first, rule which matches is used. The current implementation therefore uses a more complicated updating cycle, as we shall see is section 5.1.3.

```
    [ setRec(w^pdu,CDU),
      incr_set(next_du_id,DU),
      setRec(w^cdu, record([tognd=record([scp=stackset([]),
                                          obl=stackset([]),
                                          dh=stackset([]),
                                          opt=stack([])]),
                            id=DU])),
      pushRec(w^udus,DU) ]
  ).
```

The conditions just retrieve the value of CDU, and the first effect copies this value to PDU. The next effect is a new operation (specific to the EDIS implementation at the moment) which increments the value of a global variable representing the DU number, sets the global to be the new value, and actually sets its second argument to be the 'IS internal' representation of this value (DU2, DU3, and so on). The next effect sets up an empty CDU with the new ID, and finally the ID number of CDU is also pushed into UDUs. CDU is now ready to be filled with the contents of the utterance as represented in `latest_moves`, and the urule which fires in this context is:

```
urule( doInfoReqW, ruletype3,
   [ latest_speaker: val(w),
     latest_moves: in(Move),
     Move:valRec(pred,inforeq) ],
   [ incr_set(update_cycles,_),
     incr_set(next_dh_id,HID),
     next_du_name(ID),
     pushRec(w^cdu^tognd^dh,record([atype=Move,
                                    id=HID ])),
     pushRec(w^cdu^tognd^obl,record([pred=answer,
                                     dp=c,
                                     args=stackset([record([item=HID])])])),
     pushRec(w^gnd^obl,record([pred=uact,
                               dp=c,
                               args=stackset([record([item=ID])])])) ]
  ).
```

Based on the template rules in [CLM+99] (p.36), the conditions state that the latest speaker should be W, and that there should be a move in `latest_moves` whose predicate is `inforeq`. The first three operations handle bookkeeping of the system variables which deal with IDs and update cycles, and so the rule has three main effects; the dialogue history in CDU is updated to contain a representation of the `inforeq`, the fact that C has an obligation

to answer the question is pushed into OBL in CDU, and C's obligation to perform an understanding act is pushed directly into ground.

Note that one of the bookkeeping operations carried out by the urules in steps 2 and 3 is to increment the global variable `update_cycles` every time a rule pushes new information into the IS. This global is subsequently used to see whether there is new information in the IS which needs to be checked (as part of the implementation of step 4 in (5.4). See section 5.1.3 below for a full account of how the implementation uses this information.

To recap, given some initial intentions, we have shown how the system produces the first utterance in (5.2), and how the update rules produce the IS which results from this utterance. We can now look at the Caller's response:

(5.5) W[1]: How can I help?
C[2]: A route please

Although the intention to get a route is present in the ISs we have been using, this is really just for completeness as it is not actually used to produce the next utterance. It is necessary to explicitly set `latest_moves` and run the update cycle, and some simple macros have been written to do this. The relevant one at this point represents the caller's 'getroute' move:

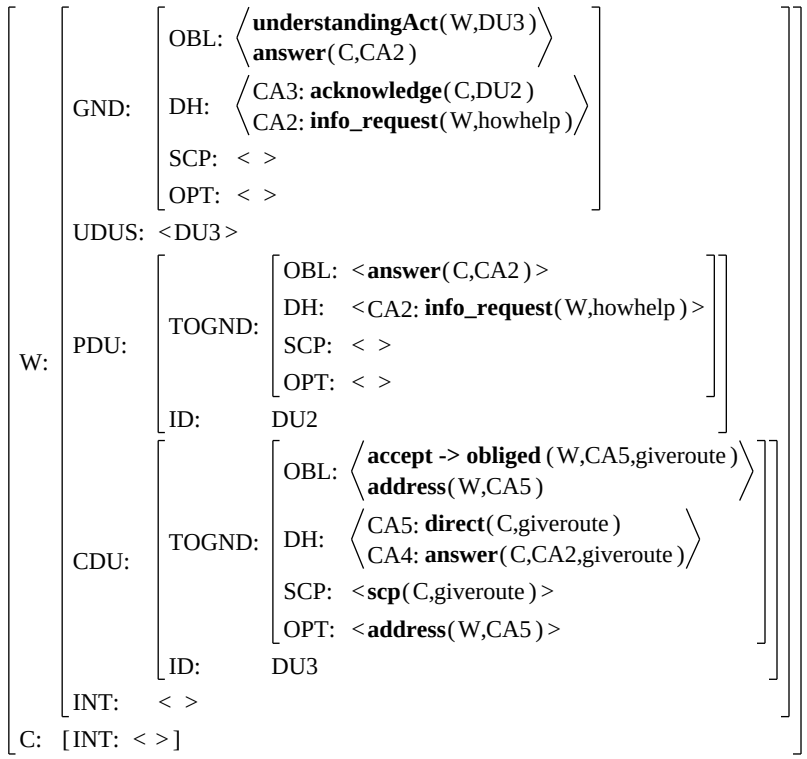
```
grmove :-
   set(latest_moves,stackset([
                     record([pred=ack,
                             dp=c,
                             args=stackset([record([item='DU2'])])]),
                     record([pred=answer,
                             dp=c,
                             args=stackset([record([item='CA2']),
                                            record([item=giveroute])])]),
                     record([pred=direct,
                             dp=c,
                             args=stackset([record([item=giveroute])])])])),
   set(latest_speaker,c),
   operation(popRec(c^int)),
   update.
```

There is a version of this macro which prints the string representing the utterance, but the important issues concern the IS alterations. Note that the utterance of [2] in (5.5) is

assumed to constitute three moves; an acknowledgement, an answer, and a directive. The acknowledgement of the previous utterance is implicit, of course, but the caller is obliged to provide it. Also implicit, in some sense, is the fact that utterance [2] provides an answer to utterance [1]; it should be possible to derive this information automatically (perhaps along the lines suggested by Jörn Kreutel), but for the moment we explicitly include it. Finally, we assume that the core act is a directive.

As shown, the 'getroute' macro runs the update cycle, and this will map (5.3) into (5.6):

(5.6)



In all, six update rules have applied in the mapping. Firstly, as before, the rule for creating a new DU has operated, copying CDU to PDU in the process. Next, as the first element in `latest_moves` is the acknowledge act, step 2 in (5.4) is implemented using the following update rule:

```
urule( doBGActsUsr, ruletype2,
       [ latest_speaker:val(c),
         latest_moves: in(Move),
         Move:valRec(pred,ack),
         valRec(w^pdu^id,PID) ],
       [ incr_set(next_dh_id,ID),
```

```
        incr_set(update_cycles,_),
        pePathRec(w^gnd,w^pdu^tognd),
        pushRec(w^gnd^dh,
                 record([atype=Move,
                         id=ID])),
        delRec(w^udus,PID)]
    ).
```

Apart from the bookkeeping, this rule is just checking to see whether `latest_moves` contains an acknowledgement by C, and if there is, it first merges PDU with the common ground using `pePathRec`. Next the acknowledgment is recorded in ground, and the ID of PDU is removed from UDUS as the information is now, of course, grounded.

Step 3 (dealing with acts other than grounding acts) now applies, and the update rule which handles answers is:

```
urule( doAnswerC, ruletype3,
       [ latest_speaker: val(c),
         latest_moves: in(Move),
         Move:valRec(pred,answer),
         Move:valRec(args,stackset([record([item=_]),
                                    record([item=ANS])])) ],
       [ incr_set(update_cycles,_),
         incr_set(next_dh_id,HID),
         next_du_name(ID),
         pushRec(w^cdu^tognd^dh,record([atype=Move,
                                        id=HID ])),
         pushRec(w^cdu^tognd^scp,record([pred=scp,
                                  dp=c,
                                  args=stackset([record([item=ANS])])])),
         pushRec(w^gnd^obl,record([pred=uact,
                                  dp=w,
                                  args=stackset([record([item=ID])])])) ]
    ).
```

This basically says that, if the first thing in `latest_moves` is an `answer` act by C, record the act in CDU and also record the fact that there is a potential commitment to the proposition expressed in the answer. Finally, as usual, the need for the other participant (W in this case) to perform an understanding act is pushed into the common ground.

The rule which handles the `direct` act is:

```
urule( doDirectC, ruletype3,
     [ latest_speaker: val(c),
       latest_moves: in(Move),
       Move:valRec(pred,direct),
       Move:valRec(args,stackset([record([item=PROP])])) ],
     [ incr_set(update_cycles,_),
       incr_set(next_dh_id,HID),
       next_du_name(ID),
       pushRec(w^cdu^tognd^dh,record([atype=Move,
                                      id=HID ])),
       pushRec(w^cdu^tognd^obl,record([pred=address,
                                       dp=w,
                                       args=stackset([record([item=HID])])])),
       pushRec(w^cdu^tognd^obl,record([pred=accept_obl,
                                       dp=w,
                                  args=stackset([record([item=HID]),
                                                 record([item=PROP])])])),
       pushRec(w^cdu^tognd^opt,record([pred=address,
                                       dp=w,
                                       args=stackset([record([item=HID])])])),
       pushRec(w^gnd^obl,record([pred=uact,
                                 dp=w,
                                 args=stackset([record([item=ID])])])) ]
    ).
```

This looks to see if the first thing in `latest_moves` is a directive, and if so, it records the act in DH in CDU. It also pushes two obligations into CDU; the first is for the other participant to `address` the directive, and the second records the implication that accepting the contents of the directive will result in an obligation to perform the proposition it expresses. The fact that W has an option to address the directive is also recorded in CDU, and finally the obligation to perform an understanding act is pushed into common ground.

Step 4 now applies, effectively cycling through the updating process. However, the only update rules which are relevant deal with different kinds of updating processes, namely those to do with resolving obligations and implicational acts. Thus the algorithm actually only cycles through the last two rule types, which deal with obligation and implication resolution. The suggestion is that the update rules themselves will not give rise directly to dialogue acts, but that the information which they push into the IS must be checked to see whether it resolves any obligations or implications.

The first type of urule which is checked deals with obligation resolution, which is basically fairly simple; if there is an obligation to perform an act and a suitable act is present in the dialogue history, remove the obligation. The following rule has therefore been included in the cycle:

```
urule( obligationRes, ruletype4,
     [ inRec(w^gnd^obl,ATYPE),
       inRec(w^gnd^dh,ACTION),
       ACTION: valRec(atype,ATYPE2),
       checkATypes(ATYPE,ATYPE2)  ],
     [ delRec(w^gnd^obl,ATYPE) ]
    ).
```

This works more or less as suggested above. The only real complication is that it is necessary to check the type of the predicates to see whether action types match; so, for instance, the presence of an acknowledgment in the dialogue history constitutes an understanding act as suggested by the taxonomy in [CLM+99] (p.37). So, the rule says that if there's an action type in OBL and a dialogue action in DH whose action type matches, remove the obligation. This rule is responsible for removing the first obligation on C to acknowledge W's first utterance.

The operations for dealing with the rest of the dialogue in (5.2) are very much the same as the machinery we have seen. However, one point to note is that there is another step in the cycle which deals with implicational acts such as that introduced by directives. Before looking more closely at the update algorithm, for completeness we can conclude this subsection with a description of the implication resolution update rules and of one refinement to the obligation resolution rule.

Taking the implication resolution rules first, the following is an example:

```
urule( acceptObl, ruletype5,
     [ inRec(w^gnd^obl,record([pred=accept_obl,
                               dp=DP,
                               args=stackset([record([item=HID]),
                                              record([item=OBL])])])),
       inRec(w^gnd^dh,record([atype=record([pred=accept,
                               dp=DP,
                               args=stackset([record([item=HID])])]),
                              id=_]))],
     [ incr_set(update_cycles,_),
       pushRec(w^gnd^obl,record([pred=obliged,
                                 dp=DP,
                            args=stackset([record([item=OBL])])])),
       delRec(w^gnd^obl,record([pred=accept_obl,
                               dp=DP,
                               args=stackset([record([item=HID]),
                                              record([item=OBL])])])) ]
    ).
```

This rule handles the kind of implicational act introduced by directives. Its conditions are looking for the appropriate act type in OBL and for a suitable `accept` act in the DH. If such a match is found, the obligation representing the consequent is pushed into ground and the implicational act itself is removed from OBL. Note that the presence of an `accept` act is required; this involves an adaption of PTT as suggested in [CLM+99]. There, the theory does not represent `accept` acts directly in ground. In order to discharge the implication, it is necessary to have something suitable in the IS, and so it is assumed here that the intention to `accept` gives rise directly to an `accept` act in the IS (cf. the annotation in [CLM+99] (p.107) in which the intention to accept the directive appears).

The complication to the obligation resolution rule deals with the final statement in [CLM+99] (p.37). This suggests that a `check` act with an accompanying `agree` have the type of an `answer`; at the moment, this is handled using another obligation resolution urule:

```
urule( checkAgree, ruletype4,
     [ inRec(w^gnd^obl,record([pred=answer,
                               dp=DP,
                               args=stackset([record([item=ACT])])])),
       inRec(w^gnd^dh,record([atype=record([pred=agree,
                                    dp=DP,
                                    args=stackset([record([item=ACT])])]),
                              id=_])),
       inRec(w^gnd^dh,record([atype=record([pred=check,
                                    dp=_,
                                    args=_]),
                              id=ACT])) ],
     [ delRec(w^gnd^obl,record([pred=answer,
                               dp=DP,
                               args=stackset([record([item=ACT])])])) ]
    ).
```

The conditions here are looking for an obligation to `answer` and for a suitable `agree` and `check` in DH. If the conditions are met, the obligation is discharged.

5.1.3 The Update Algorithm

The update algorithm is based directly on the four-step process suggested in (5.4) above, as we have noted, although the fourth step is actually an indication that the process should cycle. However, as we also noted, it was necessary to include two extra steps, and the current cycle therefore actually has five separate 'stages', which are implemented using the 'rule type' slot. The update engine used for the P-T model is:

```
update :-
  urule_engine([ruletype1,ruletype2,ruletype3]),
  print_state, !.
```

This is the same as the default except for the addition of the rule type specifications as an argument to `update`. The definition of `urule_engine` is:

```
urule_engine([]) :- val(update_cycles,UD),
                    update_cycle(UD).

urule_engine([Type|Types]) :-
        apply_all_rules_of_type(Type),
        urule_engine(Types).
```

Each rule type is tried in turn, and all rules of each type are checked. The end check, when all the types have been tried, calls the `update_cycle` predicate which checks the last two rule types, as we shall see. The `apply_all_rules_of_type` is actually quite close to the original `urule_engine` definition:

```
apply_all_rules_of_type(Type) :-
        urule(Rule, Type, Preconds, Effects ),
        check_preconds( Preconds ),
        print_rule( Rule, Effects ),
        apply_effects( Effects ), fail.
apply_all_rules_of_type(_).
```

The significant change is the forced failure which ensures that every rule is tried.

The update cycle decrements the `update_cycles` variable, trying the obligation and implication resolution rule types:

```
update_cycle(0).
update_cycle(Num) :- Num2 is Num - 1,
                     set(update_cycles,Num2),
                     urule_engine([ruletype4,ruletype5]).
```

5.2 TrindiKit Issues

The EDIS implementation required a number of (small) extensions to the DME machinery. Apart from a few bookkeeping global variables, the following were added:

```
operation(incr_set(next_dh_id,Name)) :-
     val(next_dh_id,V1),
     Val is V1 + 1,
     set(next_dh_id,Val),
     number_chars(Val,NChars),
     append([67,65],NChars,Chars),
     atom_chars(Name,Chars).
```

This allows counters to be incremented. A typical use of such a counter is in providing an ID number for an element in the dialogue history. Another new definition is:

```
operation(next_du_name(Name)) :-
     val(next_du_id,Num),
     number_chars(Num,NChars),
     append([68,85],NChars,Chars),
     atom_chars(Name,Chars).
```

Rather than set the name of the new DU, this operation just retrieves it. We also use a new condition (used when checking that dialogue acts match) which checks the type of predicates:

```
condition(predType(PRED,PREDTYPE)) :-
    ptype(PRED,PREDTYPE).

ptype(ack,uact).
ptype(accept,address).
ptype(Type,Type).
```

The `predType` condition associates dialogue act predicates with their types as specified in [CLM+99] on page 37. Thus, for instance, an acknowledgment is defined to be of type `uact` (an understanding act). One place where this information is crucial is in resolving obligations, in which case the presence of an acknowledgement in the dialogue history can be used to discharge the obligation to perform an understanding act.

Finally, we include the `checkATypes` condition, used in the obligation resolution urule, below:

```
condition(_,checkATypes(Record1,Record2),_) :-
   infostate_access:condition(Record2: valRec(pred,PRED)),
   infostate_access:condition(predType(PRED,PREDTYPE)),
   infostate_access:condition(Record1: valRec(pred,PREDTYPE)),
   infostate_access:condition(Record1: valRec(dp,DP)),
   infostate_access:condition(Record2: valRec(dp,DP)),
   infostate_access:condition(Record1: fstRec(args,ARG1)),
   infostate_access:condition(Record2: fstRec(args,ARG1)).
```

This condition is used to check whether two action types are compatible. It calls the `predType` condition discussed above to see if the types of the predicates match, and then just checks that the dialogue participant and relevant argument are identical.

Chapter 6

GoDiS

In this section we present the GoDiS system. For a more in-depth presentation of GoDiS, see [TBC+99]. GoDiS is an experimental dialogue system built using the TrindiKit. It uses fairly simple algorithms for the control, update and selection modules, keyword-based interpretation and template-based generation. GoDiS is currently able to handle simple grounding phenomena as well as *question accommodation*, allowing users to answer unasked but salient questions.

6.1 System Description

6.1.1 Information state

GoDiS uses an information state type based on the "dialogue gameboard" introduced by Ginzburg [Gin98], including the notion of QUD (Questions Under Discussion). The main division in the information state is between information which is private to the agent and that which is shared between the dialogue participants. What we mean by shared information here is that which has been explicitly established during the conversation (akin to what Lewis in [Lew79] called the "conversational scoreboard").

The private BEL field contains propositions that the system holds to be true. The AGENDA field contains the short term goals or obligations that the agent has, i.e. what the agent is going to do next. For example, if the other dialogue participant raises a question, then the agent will normally put an action on the agenda to respond to the question. The SHARED field is divided into three subfields. One subfield is a set of propositions which the agent

assumes for the sake of the conversation. The second subfield is for a stack of questions under discussion (QUD). These are questions that have been raised and are currently under discussion in the dialogue. In real human-human dialogues, dialogue participants can address questions that have not been explicitly raised in the dialogue. However, it is important that a question be available to the agent who is to interpret it because the utterance may be elliptical. An example from a travel agency dialogue[1] is shown in (6.1).

(6.1) J: what month do you want to go

P: well around 3rd 4th april / some time there

P: as cheap as possible

The strategy we adopt for interpreting elliptical utterances is to think of them as short answers (in the sense of Ginzburg [Gin96a, Gin96b, Gin98]) to questions on QUD. A suitable question here is *What kind of price does C want for the ticket?*. This question is not under discussion at the point when the customer says "as cheap as possible", but it can be figured out since J knows that "as cheap as possible" this is a relevant question. In fact it will be a question which J has as an action in his plan to raise. On our analysis it is this fact which enables J to interpret the ellipsis. He finds the matching question on his plan, accommodates by placing it on QUD and then continues with the integration of the information expressed by *as cheap as possible* as normal. The final SHARED field contains information about the latest move (speaker, move type and content).

The PLAN field is a list of dialogue actions that the agent wishes to carry out. The plan can be changed during the course of the conversation. For example, if a travel agent discovers that his customer wishes to get information about a flight he will adopt a plan to ask her where she wants to go, when she wants to go, what price class she wants and so on. In cases where the customer does not state her errand explicitly, but rather answers some question(s) (e.g. about destination and means of transport), the agent must infer what the task is. We call this process *task accommodation*, and it is closely related to question accommodation.

We have also included a field TMP that mirrors the shared fields. This field keeps track of shared information that has not yet been grounded, i.e. confirmed as having been understood by the other dialogue participant. In this way it is easy to delete information which the agent has optimistically assumed to have become shared if it should turn out that the other dialogue participant does not understand or accept it. A variant of the system (with a few different update rules) allows a system to pursue a *cautious* rather than *optimistic* strategy with respect to grounding. In this version, information will at first

[1] This dialogue has been translated from Swedish. It was collected by the University of Lund.

only be placed on TMP until it has been acknowledged by the other dialogue participant whereupon it can be moved from TMP to the appropriate shared field.

A sample information state, resulting from the dialogue in (6.2), is shown in (6.3).

(6.2)
```
Sys:  Welcome to the travel agency!
Usr:  flights to paris
Sys:  What city do you want to go from?
```

(6.3)
$$\left[\begin{array}{lll} \text{PRIVATE} & = & \left[\begin{array}{lll} \text{BEL} & = & \{\} \\ \text{AGENDA} & = & \langle\rangle \\ \text{PLAN} & = & \left\langle\begin{array}{l} \text{RAISE}(\text{R}\hat{}(\text{RETURN}{=}\text{R})), \\ \text{RAISE}(\text{M}\hat{}(\text{MONTH}{=}\text{M})), \\ \text{RAISE}(\text{C}\hat{}(\text{CLASS}{=}\text{C})), \\ \text{RESPOND}(\text{P}\hat{}(\text{PRICE}{=}\text{P})) \end{array}\right\rangle \\ \text{TMP} & = & \text{(same as SHARED)} \end{array}\right] \\ \text{SHARED} & = & \left[\begin{array}{lll} \text{BEL} & = & \{(\text{TO}{=}\text{PARIS}),(\text{HOW}{=}\text{PLANE})\} \\ \text{QUD} & = & \langle \text{X}\hat{}(\text{FROM}{=}\text{X})\rangle \\ \text{LM} & = & \text{ASK}(\text{SYS},\text{Y}\hat{}(\text{FROM}{=}\text{Y})) \end{array}\right] \end{array}\right]$$

The user utterance is seen as answering two questions; however, no task or plan has yet been established and no questions have been raised. To be able to integrate the utterance, the system must find a task associated with a plan which includes the raising of questions which match the answers given. Once the task (in this case getting price information about a trip) has been accommodated and the plan entered into the PLAN field, the questions can be accommodated and the answers integrated. As a consequence of this process, the information state now contains a plan which guides the system behavior.

6.1.2 Dialogue moves, update rules, and modules

GoDiS is implemented using 7 dialogue move types (ask, answer, request-repeat, repeat, greet, quit and thank) and 40 update rules, whereof 31 are used by the update module and 9 by the selection module. These two modules form the GoDIS DME. There are currently 8 rule classes in GoDiS: grounding, integrate, database, accommodate, manage_plan, refill, store and select.

Given the kind of information state illustrated described above, we can provide update rules which accommodate questions. A formalization of the **accommodate_question** move is given in (6.4). When interpreting the latest utterance by the other participant, the system

makes the assumption that it was a **reply** move with content A. This assumption requires accommodating some question Q such that A is a relevant answer to Q. The check operator "answer-to(A, Q)" is true if A is a relevant answer to Q given the current information state, according to some (possibly domain-dependent) definition of question-answer relevance.

(6.4)
$$\begin{array}{l} \text{U-RULE: } \textbf{accommodateQuestion}(Q, A) \\ \text{PRE: } \left\{ \begin{array}{l} \text{valRec(SHARED.LM, answer(usr,}A\text{))}, \\ \text{inRec(PRIVATE.PLAN, raise(}Q\text{))} \\ \text{answer-to(} A, Q \text{)} \end{array} \right. \\ \text{EFF: } \left\{ \begin{array}{l} \text{delRec(PRIVATE.PLAN, raise(}Q\text{))} \\ \text{pushRec(SHARED.QUD, } Q\text{)} \end{array} \right. \end{array}$$

6.1.3 Resources

There are three types of resource in GoDiS: lexicon, database and domain knowledge. The database and domain resources are language independent, while the lexicon is noth domain and language dependent. GoDiS has prototype resource packages for two domains (travel agency and autoroute) and two languages (swedish and english).

6.1.4 Parsing, generation, lexicon and semantics

The interpretation module in GoDiS 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 words which all are regarded as having the same meaning.

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.

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

6.2 TrindiKit Issues

GoDiS 1.0 uses the official TrindiKit version 1.1. Since GoDiS and TrindiKit were developed in parallell by the same people, they have been adapted for each other and thus there has not been any significant problems in using the TrindiKit. As an example, the GoDiS update rules could be simplified significantly by the introduction of association sets in TrindiKit. In a previous version, the latest moves were repesented as a set of records with one field describing the move and one containing a flag indicating whether the move had been integrated or not. To change the value of the flag, complicated operations had to be performed (finding the correct record in the set of records, setting the value of the integration flag, and replacing the old record with the new one). In the current version this is solved by representing the latest moves as an association set where each move is associated with a boolean value. To change the value, the `set_assoc/2` operation is used to set the polarity of the boolean associated with any specified move.

The update module algorithm uses the TrindiKit DME-ADL interpreter (introduced in TrindiKit 1.0) to impose a sufficient amount of order on the triggering of rules to avoid looping. Also, the general operation `forall(`*Cond*`,` *Op*`)` introduced in TrindiKit 1.1 is used extensively in GoDiS to simplify update rules.

6.3 System Evaluation

An evaluation of GoDiS, January 2000, in line with the Trindi tick-list developed in deliverable D1.3 reveals the following (**Qn 1** has been split into two subquestions):

Qn 1 *Is utterance interpretation sensitive to dialogue context?*

—Yes. Interpretation of an utterance is performed in the context of previous utterances. As an example, the easiest way for the user to specify a city of departure or a destination is by using the prepositions "from" and "to" – this can be done anywhere in the dialogue. A user answer simply stating "Paris" is then ambiguous between a departure and a destination reading, but the system shows dialogue context sensitivity by keeping track of what is being asked of the user.

Qn 2 *Is utterance interpretation sensitive to deictic context?*

—No. The system has no knowledge of the current date, and temporal deixis has therefore not been implemented.

Qn 3 *Can the system deal with answers to questions that give more information than was requested?*

—Yes. The user can perfectly well say something like:

(6.5) U: I would like to go from London to Paris in April by plane

GoDiS will recognise the subphrases "from London", "to Paris", "in April", and "plane", and extract the appropriate information, just as if the same answers were given one by one to specific questions posed by the system.

Qn 4 *Can the system deal with answers to questions that give less information than was actually requested?*

—Yes. The information actually provided by the user will be integrated by the system, just as in the case of the user supplying more information than was requested. GoDiS will then repeat the question that was not answered.

Qn 5 *Can the system deal with answers to questions that give less information than was actually requested?*

—No. Different levels of specificity are not implemented in the ontology.

Qn 6 *Can the system deal with ambiguous designators?*

—No. That is, it is not possible to make use of ambiguous designators.

Qn 7 *Can the system deal with negatively specified information*

—No. Negation is not, yet, part of GoDiS. If a user replies "Not by boat" to the question "How do you want to travel?", GoDiS recognises the subphrase "boat", and assumes that the journey in question is to be undertaken by boat.

Qn 8 *Can the system deal with no answer to a question at all?*

—? GoDiS waits until the user has typed an answer and hit the enter key. If enter is pressed without anything having been typed, GoDiS repeats its question.

Qn 9 *Can the system deal with noisy input?*

—Yes. If a nonsense phrase is given by the user, GoDiS replies: "I didn't understand that. Please rephrase". It can also be noted in this context that GoDiS makes a distinction between understanding and relevance. If the user types something that GoDiS is able to understand but does not consider relevant, GoDiS instead asks: "What do you mean by that?".

Qn 10 *Can the system deal with 'help' sub-dialogues initiated by the user?*

—No.

Qn 11 *Can the system deal with 'non-help' sub-dialogues initiated by the user?*

—Yes, to a very limited extent. At any time during the dialogue the user can ask if a visa is needed, as in for example:

(6.6) S: What city do you want to go to?

(6.7) U: Do I need a visa for the USA?

(6.8) S: You don't need any visa there

Qn 12 *Does the system only ask appropriate follow-up questions?*

—Yes, but it should be added that the various scenarios permitted by the system all involve the same kind and amount of information; for example, journeys by plane, train, and boat all involve a choice of class.

Qn 13 *Can the system deal with inconsistent information?*

—Yes. That is, the system detects inconsistencies, but is not able to do any belief revision based on contradictory information. An example is the following where a user tries to re-answer a question which has already been given a valid answer:

(6.9) U: I'd like to go from Paris

(6.10) S: How do you want to travel?

(6.11) U: From London

(6.12) S: What do you mean by that?

Qn 14 *Can the system deal with belief revision?*

—No, the system does not provide such facilities.

Qn 15 *Can the system deal with barge-in input?*

—No, the system does not provide such facilities.

Qn 16 *Is it possible to get a system tutorial concerning the kind of information the system can provide and what the constraints on input are?*

—No, not in the current version.

Qn 17 *Does the system check its understanding of the user's utterances?*

—No, since GoDiS currently uses text input this is not needed. It will however be necessary when using speech.

Bibliography

[BB98] Patrick Blackburn and Johan Bos. Representation and Inference for Natural Language. A First Course in Computational Semantics. Draft available at URL: `http://www.coli.uni-sb.de/~bos/comsem/`, July 1998.

[BBKdN99] Patrick Blackburn, Johan Bos, Michael Kohlhase, and Hans de Nivelle. Inference and Computational Semantics. In H.C. Bunt and E.G.C. Thijsse, editors, *Third International Workshop on Computational Semantics (IWCS-3)*, pages 5–19, 1999. Computational Linguistics, Tilburg University.

[BBL+99] Peter Bohlin, Johan Bos, Staffan Larsson, Ian Lewin, Colin Matheson, and David Milward. Survey of existing interactive systems. Technical Report Deliverable D1.3, Trindi, 1999.

[BMM+94] Johan Bos, Elsbeth Mastenbroek, Scott McGlashan, Sebastian Millies, and Manfred Pinkal. A Compositional DRS-based Formalism for NLP Applications. In Harry Bunt, Reinhard Muskens, and Gerrit Rentier, editors, *International Workshop on Computational Semantics*, 1994. University of Tilburg.

[CLM+99] R. Cooper, S. Larsson, C. Matheson, M. Poesio, and D. Traum. Coding Instructional Dialogue for Information States. Technical Report Deliverable D1.1, Trindi, 1999.

[FK99] A. Franke and M. Kohlhase. System description: Mathweb, an agent-based communication layer for distributed automated theorem proving. In *16th International Conference on Automated Deduction CADE-16*, 1999.

[Gin96a] J. Ginzburg. Dynamics and the semantics of dialogue. In *Logic, Language and Computation, Vol. 1*. 1996.

[Gin96b] J. Ginzburg. Interrogatives: Questions, facts and dialogue. In *The Handbook of Contemporary Semantic Theory*. Blackwell, Oxford, 1996.

[Gin98] J. Ginzburg. Clarifying utterances. In J. Hulstijn and A. Niholt, editors, *Proc. of the Twente Workshop on the Formal Semantics and Pragmatics of*

Dialogues, pages 11–30, Enschede, 1998. Universiteit Twente, Faculteit Informatica.

[KR93] Hans Kamp and Uwe Reyle. *From Discourse to Logic*. Kluwer, Dordrecht, 1993.

[LBBT99] Staffan Larsson, Peter Bohlin, Johan Bos, and David Traum. TRINDIKIT 1.0 Manual. Technical Report Deliverable D2.2, Trindi, 1999.

[Lew79] D. K. Lewis. Scorekeeping in a language game. *Journal of Philosophical Logic*, 8:339–359, 1979.

[Mil95] G. Miller. WordNet: A lexical database for English. *Communications of the ACM*, 38(11):39–41, 1995.

[MTP00] Colin Matheson, David Traum, and Massimo Poesio. Modelling grounding and discourse obligations using update rules. In *Proceedings of NAACL 2000*, April 2000.

[Mus96] Reinhard Muskens. Combining montague semantics and discourse representation. *Linguistics and Philosophy*, 19:143–186, 1996.

[Poe00] Massimo Poesio. Underspecified Interpretation. Technical Report Deliverable D5.1, Trindi, 2000.

[PT98] Massimo Poesio and David Traum. Towards an axiomatisation of dialogue acts. In J. Hulstijn and A. Nijholt, editors, *Proceedings of the Twente Workshop on the Formal Semantics and Pragmatics of Dialogues*, pages 207–222, Enschede, Universiteit Twente, Faculteit Informatica, 1998.

[TBC+99] David Traum, Johan Bos, Robin Cooper, Staffan Larsson, Ian Lewin, Colin Matheson, and Massimo Poesio. A model of dialogue moves and information state revision. Technical Report Deliverable D2.1, Trindi, 1999.

[VdS92] Rob A. Van der Sandt. Presupposition Projection as Anaphora Resolution. *Journal of Semantics*, 9:333–377, 1992.