DIPPER: Description and Formalisation of an Information-State Update Dialogue System Architecture

Johan Bos, Ewan Klein, Oliver Lemon, Tetsushi Oka
ICCS, School of Informatics
University of Edinburgh
2 Buccleuch Place, Edinburgh EH8 9LW
Scotland, United Kingdom
{jbos,ewan,olemon,okat}@inf.ed.ac.uk

Abstract

The DIPPER architecture is a collection of software agents for prototyping spoken dialogue systems. Implemented on top of the Open Agent Architecture (OAA), it comprises agents for speech input and output, dialogue management, and further supporting agents. We define a formal syntax and semantics for the DIPPER information state update language. The language is independent of particular programming languages, and incorporates procedural attachments for access to external resources using OAA.

1 Introduction

Spoken dialogue systems are complex frameworks, involving the integration of speech recognition, speech synthesis, natural language understanding and generation, dialogue management, and interaction with domain-specific applications. These components might be written in different programming languages or running on different platforms. Furthermore, with current developments in speech technology, many components for a dialogue system can be obtained "off-the-shelf", particularly those involving speech recognition and speech synthesis, and to a lesser extent those for parsing and generation. The overall behaviour of a dialogue system is controlled by the dialogue management component, where interaction between the different components is managed in a flexible way. Allowing for plug-and-play and easy adaptation to new domains is a challenging task for dialogue system architectures.

This paper presents DIPPER, an architecture tailored for prototyping spoken dialogue systems, based on the Open Agent Architecture (OAA). Although DIPPER supports many off-the-shelf components useful for spoken dialogue systems, it comes with its own dialogue management component, based on the information-state approach to dialogue modelling (Traum et al., 1999; Larsson and Traum, 2000).

The TrindiKit (Larsson et al., 1999; Larsson, 2002) is regarded as the first implementation of the information-state approach. However impressive it is, on many occasions the TrindiKit tends to give the impression of a "Rube Goldberg" machine for what is a relatively straightforward task: updating the information state of the dialogue with the help of declaratively stated update rules. What should be a transparent operation is often obscured by the complexity of the TrindiKit framework. The dialogue management component of DIPPER borrows many of the core ideas of the TrindiKit, but is stripped down to the essentials, uses a revised update language (independent of Prolog), and is more tightly integrated with OAA. We argue that the resulting formalism offers several advantages for developing flexible spoken dialogue systems.

We will first introduce OAA and DIPPER agents for building spoken dialogue systems, and explain how dialogue management interfaces with components in a flexible way (Section 2). Then we review the information-state approach to dialogue modelling, introduce the DIPPER update language (Sec-

tion 3), and compare it to the TrindiKit (Section 4). Finally, we list some practical results obtained using the DIPPER framework (Section 5).

2 The DIPPER Environment

This section gives an overview of DIPPER. First we introduce the Open Agent Architecture, then we present the various agents that play a role in spoken dialogue systems. We focus on the dialogue move engine in particular.

2.1 The Open Agent Architecture

The Open Agent Architecture, OAA for short, is a framework for integrating several software agents, possibly coded in different programming languages (C/C++, Java, Prolog) and running on different platforms (Unix, Linux, Windows), in a distributed environment (Martin et al., 1999). Because dialogue systems are typically built out of a set of independent components performing particular tasks (where in many cases some of them are "out-of-the-box" packages, such as speech recognition or speech synthesis), the OAA framework forms an ideal medium to allow easy integration of software agents for dialogue systems in a prototyping development environment.

The term "agent" within OAA refers to a software process meeting the conventions of the OAA framework. Basically, this means providing services to other agents in a particular form, using the Interagent Communication Language (ICL). Within the community of agents, service requests can be submitted to the "facilitator". This is a special agent with knowledge of available agents and their capabilities. It mediates all interactions between the agents involved in submitting and fulfilling a request.

A prototypical spoken dialogue system built on top of OAA consists of an agent for speech recognition, an agent for dialogue management, an agent for speech synthesis, and several supporting agents for specific tasks such as parsing, semantic interpretation, and generation. A distributed agent architecture allows the implementation of flexible and adaptable dialogue systems, where individual agents can easily be added (or substituted by others) to extend functionality of the overall system. It also allows the integration of multi-modal input or output in a straightforward way.

The current collection of DIPPER agents consists of the following: (1) agents for input/output modalities, (2) agents for the dialogue move engine, and (3) supporting agents. We will describe the functionality of the DIPPER agents in the remainder of this section in terms of the services they provide. We will use the OAA term "solvable" to describe the services offered by agents. The solvables of an agent are registered with the facilitator, and are implemented by function calls (in C++ and Java) or predicate definitions (in Prolog) by the agents that provide them. We will use + and - in front of arguments to indicate passing or returning values.

2.2 Input/Output Agents

DIPPER supports agents for Nuance speech recognition software (www.nuance.com) by providing wrappers written in C++ or Java. The speech recognition agent can be used in two different modes: continuous speech recognition, calling the solvable `apply_effects(+Effects)` and thereby updating the information state of the dialogue (see Section 3); and in callback mode, where the solvable `recognize(+Grammar,+Time,-Input)` starts recognition using the speech grammar `Grammar` and returns `Input`, within a time specified by `Time`. The value of `Input` is determined by the grammar used as language model for speech recognition. Callback mode makes it easy to plug in new grammars during different stages of the dialogue so as to increase speech recognition performance.

On the output side, DIPPER provides agents for the speech synthesisers Festival (Taylor et al., 1998) and rVoice (www.rhetorical.com). The solvables for these output agents are `text2speech(+Text)` and `sable2speech(+Sable)`. The latter can be used to synthesise strings marked up in SABLE, an XML schema for text-to-speech (Sproat et al., 1998). A further agent is available to control Greta, a three-dimensional talking head (Pasquariello and Pelachaud, 2001).

2.3 Dialogue Management Agents

The dialogue manager forms the heart of a dialogue system, reading the input modalities, updating the current state of the dialogue, deciding what to do next, and generating output. In terms of interaction with other agents, it is the most complex component. In fact, the DIPPER dialogue manager is implemented as two cooperating OAA agents: the dialogue move engine (DME), and a DME server.

The DME does the real work by dealing with input from other agents (normally the input modalities, such as speech recognition), updating its internal state, and calling other agents (normally the output modalities, such as speech synthesis). The solvables of the DME are `check_conds(+Conditions)` and `apply_effects(+Effects)`. The former is used for other agents to check the current state of the dialogue, the latter is used to change the state (for instance by integrating results of speech recognition). (At this point these services might seem fairly abstract, but they will be made more concrete in Section 3.)

The DME server is an agent mediating between the DME and other agents. It collects requests submitted by the DME, waits for the results, and posts these back to the DME. The DME server enables the DME to manage information-state updates in an asynchronous way. Because the DME server is implemented as a multi-threaded system, it is able to cope with multiple requests at the same time. The solvable that the DME server supports is `dme(+Call,+Effects)`. On receiving this call, the DME server posts the solvable `Call` to the facilitator, waits for the result, and subsequently returns the results to the DME using its solveable `apply_effects(+Effects)`.

Let's illustrate this with an example. Suppose that the dialogue system just asked the user a yes-no question, and is ready to accept a yes-no answer. It will need to tell the speech recognition agent to load the grammar for yes/no-answers and return a result (say, within 7 seconds) at the `is^input` field of the dialogue state (see Section 3 for more details). This is done by posting the solvable:

```
dme(recognize('.YesNo',7,X),
    [set(is^input,X)])
```

To summarise the functionality of the DME, there are three ways it is able to communicate with other agents in a dialogue system: (1) agents can call the DME agent directly, using `check_conds(+Conditions)` and `apply_effects(+Effects)`; (2) the DME agent can call other agents directly, in particular if it is not interested in the results of those requests; (3) the DME agent can use the DME server as a mediating agent, normally when the results are needed for updating the information state of the DME.

The advantage of this architecture is the flexibility imposed by it, while at the same time allowing asynchronous interaction of the input/output and supporting agents with the dialogue move engine.

2.4 Supporting Agents

OAA itself comes with agents for parsing and generating based on the Gemini system (Dowding et al., 1993). DIPPER provides a further set of agents to deal with natural language understanding, based on Discourse Representation Theory (Kamp and Reyle, 1993). There is an ambiguity resolution agent that resolves underspecified DRSs into fully resolved DRSs, and there is an inference agent that checks consistency of DRSs, using standard first-order theorem proving techniques, including the theorem prover SPASS (Weidenbach et al., 1999) and the model builder MACE (McCune, 1998). DIPPER also includes a high-level dialogue planning component using O-Plan (Currie and Tate, 1991) which can be used to build domain-specific content plans.

3 The Information-state Approach

In this section we will briefly review the information-state approach and then introduce a revised version of the TrindiKit's dialogue move engine (Traum et al., 1999), including a new update language for information states.

3.1 Some History

Traditional approaches to dialogue modelling can roughly be classified as dialogue state approaches or plan-based approaches. In the former the dialogue dynamics are specified by a set of dialogue states, each state representing the results of performing a dialogue move in some previous state. The latter are used for more complex tasks requiring flex-

ible dialogue behaviour. The information-state approach (Traum et al., 1999) is intended to combine the strengths of each paradigm, using aspects of dialogue state as well as the potential to include detailed semantic representations and notions of obligation, commitment, beliefs, and plans.

The information-state approach allows a declarative representation of dialogue modelling. It is characterised by the following components:

1. a specification of the contents of the information state of the dialogue,
2. the datatypes used to structure the information state,
3. a set of update rules covering the dynamic changes of the information state, and
4. a control strategy for information state updates.

As mentioned earlier, the first fully fledged implementation of the information-state approach was the TrindiKit (Larsson et al., 1999). Written in Prolog, the TrindiKit implements dialogue systems by defining information states, update and selection rules, and control algorithms governing the rules to be applied to the information state. The DIPPER dialogue move engine builds on the TrindiKit by adopting its record structure and datatypes to define information states. However, there are some fundamental differences, the most important being that there are no update algorithms in the DIPPER DME, there is no separation between update and selection rules, and the update rules are abstracted away from Prolog. We will consider these differences in more detail in Section 4.

3.2 Specifying Information States

The information state of a dialogue "represents the information necessary to distinguish it from other dialogues, representing the cumulative additions from previous actions in the dialogue, and motivating future action" (Traum et al., 1999). The term *information state* is very abstract, and concepts such as mental model, discourse context, state of affairs, conversational score, and other variations on this theme can be seen as instances of an information state.

Like TrindiKit, DIPPER defines information states using a rich set of datatypes, including records, stacks, and queues.[1] The TrindiKit allows developers to define specific information states, tailored to a particular theory or a special task. An information state is normally defined as a recursive structure of the form *Name:Type*, where *Name* is an identifier, and *Type* a datatype. Here is a simple example:

Example 1 Information State Definition

```
is:record([grammar:atomic,
           input:queue(atomic),
           sem:stack(record([int:atomic,
                             context:drs]))]).
```

This example defines an information state as a record named `is`, consisting of the fields `grammar`, `input`, and `sem`. The field `input` is itself defined as a queue of atomic typed structures, and the field `sem` is defined as a stack of records containing the fields `int` and `context`.

As in the TrindiKit, DIPPER uses a system of references to anchor conditions and actions in the information state. Each record consists of a set of fields. Following the convention of the TrindiKit, we use the operator `^`, where `a^b` refers to the value of field `b` in record `a`, and call these *paths*. For instance, the path `is^input` in the above example refers to a queue of terms of type *atomic*. Note that paths can be arbitrarily long and may be used in conjunction with functions defined in the update language, which we will introduce in the next section.

3.3 The DIPPER Update Language

We will present the DIPPER update language here in a rather informal way, merely by using examples. (The reader is referred to the appendix for a precise definition of the update language.) The update language defines the core of the formalism underlying the information state approach: the update rules.

An update rule is a triple $\langle$*name, conditions, effects*$\rangle$, with *name* a rule identifier, *conditions* a set of tests on the current information state, and *effects* an ordered set of operations on the information state. Update rules specify the information state change potential in a declarative way: applying an update

[1] For the purpose of this paper, we restrict ourselves to a small number of datatypes, although the implementation supports further types including sets, ordered sets, numbers, and discourse representation structures.

rule to an information state (assuming a shared vocabulary of fields) results in a new state.

The conditions and effects of update rules are both recursively defined over *terms*. The terms allow one to refer to a specific value within the information state, either for testing a condition, or for applying an effect. There are two kinds of terms: *standard terms* and *anchored terms*. The standard terms define the data structures for the types (atomic types, queue, stack, records, and so on), whereas the anchored terms allow us to refer to sub-structures of the information state (such as `first` and `last` to refer to the first respectively last item of a queue). A particularly useful anchored term is of the form `T^f`, referring to a field `f` in a record `T`.

As we saw earlier the information state itself is a structure of type *record*. We refer to the information state object with the unique fixed name `is` (which belongs to the anchored terms). To illustrate reference of terms with respect to a certain information state, consider the following example, using the definition as given in Example 1.

Example 2 Information State

```
is: grammar: '.YesNo'
    input: <>
    sem: < int: model(...)
           drs: drs([X,Y],[...]) >
```

As defined in the Appendix, we will use the interpretation function $[\![.]\!]_s$ for (standard and anchored) terms with respect to an information state s. Now, with respect to the information state in Example 2, the value of $[\![\texttt{is\^{}grammar}]\!]_s$ denotes `'.YesNo'`, whereas the value of $[\![\texttt{grammar}]\!]_s$ denotes `grammar`, because the term is not anchored. Similarly, $[\![\texttt{top(is\^{}sem)\^{}drs}]\!]_s$ yields `drs([X,Y],[...])`. However, note that $[\![\texttt{top(sem)\^{}drs}]\!]_s$ is undefined. This term is not well-formed since `sem` is of type *atomic* and not of type *record*.

This example (and the ones that follow) illustrates the power and ease with which we can refer to specific attributes of the information state, and thereby specify the conditions and effects of update rules. The crucial property of conditions is that they must not change the content of the information state, and are only used to inspect values denoted by paths in the record defining the information state (such as checking identity of terms or whether a queue is empty or not), in order to trigger the effects of an update rule. Effects, on the other hand are responsible for changing the information state. There are two kinds of effects: operations (defined over terms), and solvables. The former include assignments of values to information state attributes and operations on datatypes such as stacks and queues. The latter are OAA-solvables that allow us to fulfil requests by supporting agents or input/output agents of the dialogue system, which is a useful way of incorporating procedural attachment using the functionality provided by OAA as described in Section 2. As a result, external actions are able to update the information state, giving the properties of an asynchronous architecture while maintaining a central unit for data processing.

3.4 A simple example

The following (extremely simple) example illustrates the DIPPER architecture and the information state update language. The example implements a "parrot", where the system simply repeats what the user says. Four OAA agents are involved: one agent for the speech recogniser, one for the synthesiser, and an agent each for the DME and the DME server. We will use the following information structure:

```
is:record([input:queue(basic),
           listening:basic,
           output:queue(basic)]).
```

That is, there are three fields: a queue containing the input of the speech recogniser (we're assuming that the objects returned by the speech recogniser are strings), an auxiliary field keeping track of whether speech recognition is active or not, and an output field for the text-to-speech synthesiser.

There are four update rules. The first rule, `timeout`, deals with the situation where the speech recognition returned 'timeout' (no speech was recognised in the given time). In that case we simply remove it from the queue.

```
urule(timeout,
      [first(is^input)=timeout],
      [dequeue(is^input)]).
```

By virtue of the second rule, `process`, we simply move the string from the input queue to the output queue. (This is just done for the sake of the example, we could have directly sent it to the synthesiser).

```
urule(process,
      [non-empty(is^input)],
      [enqueue(is^output,first(is^input)),
       dequeue(is^input)]).
```

The third rule, `synthesise`, gives the string to the synthesiser, by posting an OAA solvable. We are not interested in any result that could be yielded by the solvable, so the set of effects is empty here.

```
urule(synthesise,
      [non-empty(is^output)],
      [solve(text2speech(first(is^output)),[]),
       dequeue(is^output)]).
```

A slightly more complicated rule is `recognise`. It activates the speech recognition agent (with the grammar `'.Simple'`) when the system is currently not listening, then sets the listening flag to `yes` (to prevent application of the update rule again). The results of speech recognition will be integrated by the effects stated as the third argument of `solve`: the results will be placed in the `input` field, and the flag `listening` is set to `no` again.

```
urule(recognise,
      [is^listening=no],
      [solve(X,recognise('.Simple',10),
             [enqueue(is^input,X),
              assign(is^listening,no)]),
       assign(is^listening,yes)]).
```

Finally, we would like to make a remarks about the dynamics of effects in update rules. The effects are ordered, because the information state is updated after each single effect, and hence the order in which the effects are applied to the information state matters. Conditions in update rules, however, are not ordered.

4 Comparison with TrindiKit

Now that we have introduced the DIPPER information state update language, we are in a good position to compare DIPPER's approach to dialogue management that of the TrindiKit. We will consider the use of variables, controlling update rules, and distributed processing.

4.1 Use of Variables

The DIPPER update language is essentially a variable-free language (apart from the variables that are used in `solve/3` to return answers which are then substituted for the variable's occurrences in the effects). In the TrindiKit, Prolog variables are used for references to objects in the information state. The scope of such variables includes the conditions and effects of the update rule. The system of reference in DIPPER is functional rather than relational, which we will illustrate with two examples.

Example 3 In DIPPER, pushing the top element of stack `is^a` on another stack `is^b`, and consequently pop the first stack, the effects `[push(is^b,top(is^a)), pop(is^a)]` will be the way to achieve this. In the TrindiKit, one would need the effects `[is::fst(a,X), is::pop(a), is::push(b,X)]` to get the same result, where `X` denotes a Prolog variable.

Example 4 Given the information state structure presented at the beginning of this section, the term `assign(top(is^sem)^int,m)` picks the first record out of a stack, and refers to one of its fields (here, the field `int`). In the TrindiKit, this needs to be coded as `[is::fst(sem,X),X::set(int,m)]`, where again `X` denotes a Prolog variable.

In both examples the TrindiKit relies on Prolog unification to obtain the correct results. As a consequence, the order of conditions in the TrindiKit is crucial. Furthermore, in the TrindiKit it is common practice to use variables in the conditions to refer to values in the effects of update rules. Unification combined with Prolog's backtracking can sometimes lead to unexpected behaviour, causing errors that are difficult to debug (Burke et al., 2002). The DIPPER update language does not rely on Prolog, and therefore poses no such problems for dialogue system developers unfamiliar with Prolog.

4.2 Control in DIPPER

In contrast to the TrindiKit, which comes with a special language to define the update control algorithm, the control strategy used in DIPPER to select update rules is simple and completely determined by the update rules. Furthermore, there is no distinction between update and selection rules (used for selecting a new dialogue move to be made by the system) which the TrindiKit makes. The DIPPER update algorithm is characterised by the following pseudo-code:

```
1 WHILE running
2   deal with OAA-events;
```

```
3   IF there is a rule whose condi-
tions are satisfied by the informa-
tion state
4   THEN apply its effects;
5 ENDWHILE
```

Line 2 deals with external OAA agents requesting a service from the DME, in this case the solvable `apply_effects(+Effects)`. If there are any such requests, the information state gets updated, and the algorithm proceeds with line 3. Here we simply choose the first rule in the database whose conditions are satisfied by the information state and apply its effects to the information state (line 4). If there is no such rule, no updates take place and only an external event can change the information state. Note that the effects of at most one rule will be applied before proceeding to the end of the while-loop, ensuring that incoming OAA-events are regularly checked.

4.3 OAA Integration

Allowing OAA-solvables in the effects of update rules, a facility that the TrindiKit lacks, is an intuitive way of interfacing other components of a dialogue system (see the example update rules in Section 3.4). This feature allows components to be easily replaced by others with the same functionality, which is defined purely in terms of the OAA solvables. For instance, changing the synthesiser does not affect the dialogue management component.

The direct handle on OAA technology further allows one to implement advanced functionality for dialogue systems such as dealing with barge-in and multi-modal input. Most spoken dialogue systems exhibit a pipelined architecture with the following components: automatic speech recognition → natural language understanding → dialogue management → natural language generation → speech synthesis. Because DIPPER builds on the OAA framework, it allows developers to design asynchronous dialogue systems in a relatively straightforward way.

5 Practical Results

5.1 Prototyping

As the example in the previous section demonstrated, relatively little effort is required to build the core of a new dialogue system. First of all, the developer needs to select the OAA agents. A skeleton for a spoken dialogue system could consists of the Nuance speech recognition agent, the DME, and a synthesiser. Further work involves defining the information state, and the update rules. Once a core system has been built, it is often easy to switch to new domains, using a similar configuration as in previously implemented systems.

5.2 Debugging

A disadvantage of the information-state approach is that it makes testing and debugging of dialogue systems notoriously difficult. The more advanced applications require at least a couple of dozen update rules, and even for a relatively small set of rules developers tend to lose the overview of the intended behaviour of their system.

Formal testing is one possibility, where intended effects of update rules could be verified by future information states, or testing whether the conditions of an update rule guarantee that its effects can be applied to any information state defined over the same vocabulary. Given the formal specification of conditions and effects, an interesting topic for future research would be to apply model checking techniques to dialogue system development. Most of the model checking tools do not work on the more complex datatypes required by the information-state approach, although these probably can be translated into some kind of propositional representation.

Practically, the DIPPER environment offers a graphical user interface that assists during development (Figure 1). This GUI starts and stops the DME and keeps a history of updates. In addition, the developer is able to engage in “time-travelling”, by backtracking in the dialogue and restarting the dialogue from any point in the past.

Further functionality of the GUI includes the ‘Step’ function, which applies just one update rule before returning control to the GUI. This function is particularly helpful in verifying the intended effect of an update rule. Finally, the ‘Spy’ function displays all rules that are satisfied by the current information state.

5.3 DIPPER Prototypes

The number of successful spoken dialogue prototypes implemented using DIPPER is a convincing proof-of-concept. Applications include conversa-

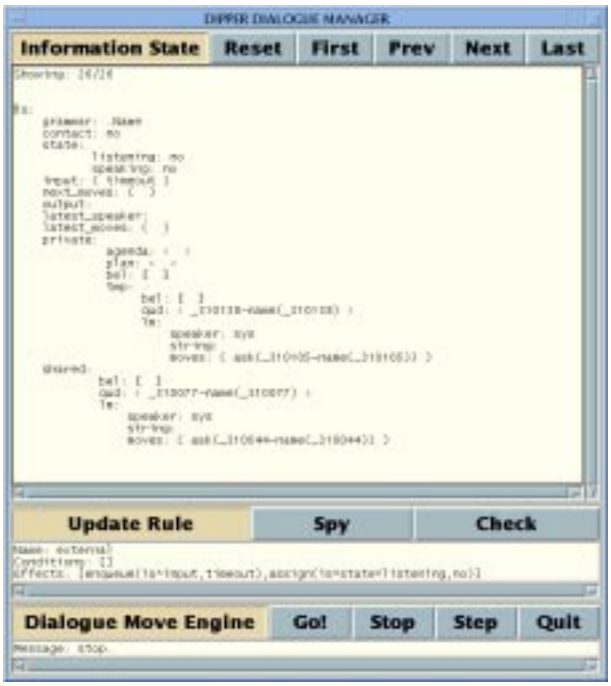

Figure 1: The Graphical User Interface of the DIPPER DME, showing the current information state, the last applied update rule, and system messages.

tion with domestic appliances, as initiated by the EU project D'Homme (Bos and Oka, 2002), explaining route descriptions to a mobile robot in a miniature town, an EPSRC-funded project (Lauria et al., 2001), and meaningful conversation with a mobile robot in the basement of our department (Theobalt et al., 2002). Currently we are working on a prototype dialogue system including the Greta three-dimensional talking head (Pasquariello and Pelachaud, 2001) as part of the EU project MagiCster.

6 Conclusion

We presented the DIPPER framework for building spoken dialogue systems, based on the information state theory of dialogue management. In comparison to TrindiKit, we showed that DIPPER provides a transparent and elegant way of declaring update rules—independent of any particular programming language, and with the ability to use arbitrary procedural attachment via OAA. The system incorporates many off-the-shelf OAA agents, which we described, as well as a variety of support agents. The DIPPER resources are available at `http://www.ltg.ed.ac.uk/dipper`.

We also presented the formal syntax and semantics of our information-state update language. Although it is up to the developer to ensure the validity of update rules, this formalisation could form the basis of implementing an interpreter that proves validity of update rules. This is an attractive task for future work, and similar directions have been suggested by (Ljunglöf, 2000; Fernández, 2003) for proving generic properties of dialogue systems.

Acknowledgements

Part of this work was supported by the EU Project MagiCster (IST 1999-29078). We thank Nuance for permission to use their software and tools.

References

J. Bos and T. Oka. 2002. An Inference-based Approach to Dialogue System Design. In *COLING 2002. Proceedings of the 19th International Conference on Computational Linguistics*, pages 113–119, Taipei.

C. Burke, L. Harper, and D. Loehr. 2002. A Dialogue Architecture for Multimodal Control of Robots. In *International CLASS Workshop on Natural, Intelligent and Effective Interaction in Multimodal Dialogue Systems.*

K. Currie and A. Tate. 1991. O-Plan: the open planning architecture. *Artificial Intelligence*, 52:49–86.

J. Dowding, M. Gawron, D. Appelt, L. Cherny, R. Moore, and D. Moran. 1993. Gemini: A natural language system for spoken language understanding. In *Proceedings of the Thirty-First Annual Meeting of the Association for Computational Linguistics.*

R. Fernández. 2003. A dynamic logic formalisation of the dialogue gameboard. In *Proceedings of the 10th Conference of the European Chapter of the ACL. Student Research Workshop*, pages 17–24, Budapest.

H. Kamp and U. Reyle. 1993. *From Discourse to Logic; An Introduction to Modeltheoretic Semantics of Natural Language, Formal Logic and DRT*. Kluwer, Dordrecht.

S. Larsson and D. Traum. 2000. Information state and dialogue management in the trindi dialogue move engine toolkit. *Natural Language Engineering*, 5(3–4):323–340.

S. Larsson, A. Berman, J. Bos, L. Grönqvist, P. Ljunglöf, and D. Traum. 1999. A model of dialogue moves

and information state revision. Technical Report D5.1, Trindi (Task Oriented Instructional Dialogue).

S. Larsson. 2002. *Issue-based Dialogue Management*. Ph.D. thesis, Goteborg University.

S. Lauria, G. Bugmann, T. Kyriacou, J.Bos, and E. Klein. 2001. Training Personal Robots Using Natural Language Instruction. *IEEE Intelligent Systems*, 16(5):38–45, Sept./Oct.

P. Ljunglöf. 2000. Formalizing the dialogue move engine. In *Götalog workshop on semantics and pragmatics of dialogue*.

D. L. Martin, A. J. Cheyer, and D. B. Moran. 1999. The open agent architecture: A framework for building distributed software systems. *Applied Artificial Intelligence*, 13.

W. McCune. 1998. Automatic Proofs and Counterexamples for Some Ortholattice Identities. *Information Processing Letters*, 65(6):285–291.

S. Pasquariello and C. Pelachaud. 2001. Greta: A simple facial animation engine. In *6th Online World Conference on Soft Computing in Industrial Appications*.

R. Sproat, A. Hunt, M. Ostendorf, P. Taylor, A. Black, and K. Lenzo. 1998. Sable: A standard for tts markup. In *ICSLP98*, pages 1719–1724.

P. A. Taylor, A. Black, and R. Caley. 1998. The architecture of the festival speech synthesis system. In *The Third ESCA Workshop in Speech Synthesis*.

C. Theobalt, J. Bos, T. Chapman, A. Espinosa-Romero, M. Fraser, G. Hayes, E. Klein, T. Oka, and R. Reeve. 2002. Talking to Godot: Dialogue with a Mobile Robot. In *Proceedings of IROS 2002*.

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

C. Weidenbach, B. Afshordel, U. Brahm, C. Cohrs, T. Engel, E. Keen, C. Theobalt, and D. Topic. 1999. System description: Spass version 1.0.0. In Harald Ganzinger, editor, *16th International Conference on Automated Deduction, CADE-16*, volume 1632 of *LNAI*, pages 314–318. Springer-Verlag, Berlin.

Appendix: Syntax and Semantics of the DIPPER Update Language

The terms of the update language refer to a specific value within the information state, either for testing a condition, or for applying an effect. There are two kinds of terms: standard terms and anchored terms.

Definition: Standard Terms.

1. All constants are standard terms of type *atomic*.
2. If $\mathrm{T}_1, \ldots, \mathrm{T}_n$ are standard terms of type τ, then $\langle \mathrm{T}_1, \ldots, \mathrm{T}_n \rangle$ is a standard term of type *stack*(τ).
3. If $\mathrm{T}_1, \ldots, \mathrm{T}_n$ are standard terms of type τ, then $(\mathrm{T}_1, \ldots \mathrm{T}_n)$ is a standard term of type *queue*(τ).
4. If $\mathrm{f}_1, \ldots, \mathrm{f}_n$ are record fields, $\mathrm{T}_1, \ldots, \mathrm{T}_n$ are terms of type $\tau_1, \ldots, \tau_n$, then $[\mathrm{f}_1{:}\mathrm{T}_1, \ldots, \mathrm{f}_n{:}\mathrm{T}_n]$ is a standard term of type *record*($\mathrm{f}_1{:}\tau_1, \ldots, \mathrm{f}_n{:}\tau_n$).
5. Standard Terms are only defined on the basis of (1)–(4).

Definition: Anchored Terms.

1. `is` is an anchored term of type *record*($\mathrm{f}_1{:}\tau_1, \ldots, \mathrm{f}_n{:}\tau_n$).
2. If T is an anchored term of type *record*($\ldots, \mathrm{f}{:}\tau, \ldots$), then T^f is an anchored term of type τ.
3. If T is an anchored term of type *queue*(τ), then `first`(T) and `last`(T) are anchored terms of type τ.
4. If T is an anchored term of type *stack*(τ), then `top`(T) is an anchored term of type τ.
5. If T is an anchored term of type *queue*(τ) or *stack*(τ), then `member`(T) is an anchored term of type τ.
6. Anchored terms are only defined on the basis of (1)–(5).

The interpretation function $[\![.]\!]_s$ for (standard and anchored) terms with respect to an information state s is defined as follows.

Definition: Reference of Terms.

1. $[\![\mathrm{T}]\!]_s$ = T iff T is a standard term.
2. $[\![\texttt{is}]\!]_s = s$.
3. $[\![\mathrm{T}\texttt{\^{}}\mathrm{f}]\!]_s$ = the value of field f in $[\![\mathrm{T}]\!]_s$.
4. $[\![\texttt{top}(\mathrm{T})]\!]_s$ = the top member of $[\![\mathrm{T}]\!]_s$ iff T is of type *stack*().

5. $[\![$`first`(T)$]\!]_s$ = the first member of $[\![$T$]\!]_s$ iff T is of type *queue*().

6. $[\![$`last`(T)$]\!]_s$ = the last member of $[\![$T$]\!]_s$ iff T is of type *queue*().

7. $[\![$`member`(T)$]\!]_s$ = a member of $[\![$T$]\!]_s$ iff T is of type *stack*() or of type *queue*().

Now we define the syntax and semantics of update rule conditions in DIPPER. For the interpretation of conditions we use a truth-conditional semantics mapping conditions to one of the values 1 ('true') or 0 ('false'), defined with the help of an interpretation function I with respect to an information state s.

Definition: Syntax of Conditions.

1. If T_1 and T_2 are (standard or anchored) terms of the same type, then T_1=T_2 and $T_1 \neq T_2$ are conditions.

2. If T is a (standard or anchored) term of type *stack*(τ), or *queue*(τ), then `empty`(T) and `non_empty`(T) are conditions.

3. Conditions are only defined on the basis of (1) and (2).

Definition: Semantics of Conditions.

1. $I_s(T_1=T_2) = 1$ iff $[\![T_1]\!]_s = [\![T_2]\!]_s$

2. $I_s(T_1 \neq T_2) = 1$ iff $[\![T_1]\!]_s \neq [\![T_2]\!]_s$

3. I_s(`empty`(T)) = 1 iff $[\![$T$]\!]_s$ denotes a stack or queue containing no elements.

4. I_s(`non_empty`(T)) = 1 iff $[\![$T$]\!]_s$ denotes a stack or queue containing at least one element.

Definition: Information State Satisfaction.
An information state s satisfies a set of conditions C iff $\forall c : c \in C \rightarrow [\![c]\!]_s = 1$.

The effects in an update rule are responsible for changing the information state. There are two kinds of effects: operations defined over terms, and solvables.

Definition: Syntax of Effects.

1. If T_1 is an anchored term of type τ and T_2 a (standard or anchored) term of type τ, then `assign`(T_1,T_2) is an effect.

2. If T_1 is an anchored term of type *stack*(τ) and T_2 a (standard or anchored) term of type τ, then `clear`(T_1), `pop`(T_1), and `push`(T_1,T_2) are effects.

3. If T_1 is an anchored term of type *queue*(τ) and T_2 a (standard or anchored) term of type τ, then `clear`(T_1), `dequeue`(T_1), and `enqueue`(T_1,T_2) are effects.

4. If the term S is an n-place OAA-solvable, T_1,...,T_n are (standard or anchored) terms, E(x) an ordered (possibly empty) set of effects with free occurrences of x, then `solve`(x,S(T_1,...,T_n),E) is an effect.

5. Effects are only defined on the basis of (1)–(4).

The semantics of the effects are defined with the help of the function U: s $\times$ E$\rightarrow$s from an information state and an effect to a new information state. (Some notational conventions: We will use the notation s[T]s' to mean that the information states s and s' are the same except for the value of $[\![$T$]\!]_s$. We will use E[t/u] to mean substituting t for u in E).

Definition: Semantics of Effects.

1. U(s,`assign`(T,T′)) = s′ if s[T]s′ and $[\![$T$]\!]_{s'} = [\![$T′$]\!]_s$.

2. U(s,`clear`(T)) = s′ if s[T]s′ and $[\![$T$]\!]_{s'} = \langle\rangle$.

3. U(s,`pop`(T)) = s′ if s[T]s′ and $[\![$T$]\!]_s = \langle t_1, t_2, \ldots, t_n\rangle$ and $[\![$T$]\!]_{s'} = \langle t_2, \ldots, t_n\rangle$.

4. U(s,`push`(T,T′)) = s′ if s[T]s′ and $[\![$T$]\!]_s = \langle t_1, \ldots, t_n\rangle$ and $[\![$T$]\!]_{s'} = \langle [\![$T′$]\!]_s, t_1, \ldots, t_n\rangle$.

5. U(s,`dequeue`(T)) = s′ if s[T]s′ and $[\![$T$]\!]_s = (t_1, t_2, \ldots, t_n)$ and $[\![$T$]\!]_{s'} = (t_2, \ldots, t_n)$.

6. U(s,`enqueue`(T,T′)) = s′ if s[T]s′ and $[\![$T$]\!]_s = (t_1, \ldots, t_n)$ and $[\![$T$]\!]_{s'} = (t_1, \ldots, t_n, [\![$T′$]\!]_s)$.

7. U(s,`solve`(x,S(T_1,...,T_n),E)) = s if for all answers a returned by solve(S($[\![T_1]\!]_s$,...,$[\![T_n]\!]_s$)) there is an s' such that the effects E[a/x] are applied to s'.

Definition: Update.
An ordered set of effects $\{e_1, \ldots, e_n\}$ are successfully applied to an information state s, resulting an information state s' if U(e_1,s)=s_1 ,..., U(e_i,s_{i-1})=s_i ,..., U(e_n,s_{n-1})=s'.