1 Introduction

Many simple dialogue systems are constructed in a more or less holistic fashion, not making clear differentiations between linguistic, dialogue, and domain components or reasoning, treating everything other than speech input and output as the "dialogue component". Such architectures allow shortcuts in the design process and fine-tuning to the particular anticipated task and dialogue interaction, which can speed up both system implementation time and run-time. However, the resulting systems are not particularly portable to other domains, tasks within the same domain, or even very robust in the face of different styles of interaction in accomplishing the task. Often, where the dialogue component is concerned, all that can be carried over into the next system is the experience gained by building such a system. Toolkits for constructing scripted dialogues, such as [32] make the construction process faster, but do not address the underlying problem of partitioning dialogue knowledge from linguistic and domain knowledge in order to reuse the same dialogue strategies.

Simply partitioning the knowledge sources is also not sufficient to achieve domain-independent reusable dialogue modules. Like a holistic system, a dialogue component (in the narrow sense, dealing strictly with context updates and decision-making about what to say) must have appropriate access to both linguistic and domain knowledge sources in order to perform appropriate dialogue actions. While there will always be a certain amount of work involved in adapting a generic dialogue module to particular linguistic processing components and domain knowledge sources and manipulators, there is still some room for generic dialogue function, abstracting away from the specific representations provided by other modules. The key is being able to represent aspects of the dialogue in a suitably abstract fashion, to allow reasoning about generalities without relying on peculiarities of interfaces to linguistic and domain modules. We maintain, agreeing with [20], that it is important to keep several different kinds of representations of an utterance available as context, in order to act appropriately in the face of meta-dialogue, such as corrections, as well as to be able to give the right kind of feedback about problems in the system's ability to interpret and act appropriately.

As an example of a simple dialogue episode which can motivate the kinds of representation we propose, consider the exchange schema in (1). In order to understand and respond to [3] properly, B must at least keep some context around of [1] and [2]. This raises the question as to how to represent this context in a compact and useful form.

(1)	[1]	A: do *X*.
	[2]	B: [does something]
	[3]	A: no, do ...

In the next section, we quickly review several structural proposals for representation of local exchanges like (1). Then in Section 3, we reconsider these proposals in the light of a suite of examples of different kinds of negative feedback. This leads us, in Section 4, to propose a representation based on considering not just the utterances themselves, but other intensional information associated with the utterances. These include, for a request produced by the user of a system: the literal request, an interpreted version, still at the level of natural language description, and a domain-specific version. For the reply, this also includes both the plan leading to its performance, as well as observed feedback. These various levels provide both a source for detecting potential or actual incoherence in dialogue, as well as serving as a source of potential repair requests. In Section 5, we illustrate these levels in action in a dialogue manager for the TRAINS-96 system [3]. Finally, we conclude with some observations of more general applicability of these levels.

2 Representations of Local Dialogue Structure

There have been several proposals for the kind of dialogue unit represented in (1), using structural terms like *adjacency pair* [27], *exchange* [31], *game* [29, 5], *IR-unit* [1] and *argumentation act* [36]. At an abstract level, we need an *IRF-unit* which can contain the three moves or acts: Initiative, Response, and Feedback, as indicated in (2).

(2) [1] Initiative: Request(Act) [Instruct]
[2] Response: Do(Act)
[3] Feedback: Eval [+ Counter-Request(Act')]

There are several ways in which this unit could be structured. In Figure 1 we show several proposed structures for this or similar units for questions. (A) shows a flat structure containing all three acts, as proposed by [31]. Some authors prefer to allow only binary branching units, which leads to structures (B) through (D). (B) was proposed by [38] (though with the unit names Solicit-Give and Give-Acknowledge), and is also used by [20]. (C) and (D) were both proposed in [29], the former for information-seeking questions, and the latter for exam-questions. (E) shows a finite automaton which could be induced from these structures, allowing multiple rejections and counter-requests before a final acceptance.

There may be different motivations for these different types of structures, but for the present purposes, we will consider them strictly in terms of what kind of context is provided for the antecedent of the utterance [3]. In particular, what is the utterance of "no" referring to: A's initial utterance [1], B's reaction in [2], or some other construct? Of course such examinations really require both the structure itself as well as an algorithm for traversing the structure and deciding on a referent. For the present purposes, we will consider a default algorithm, in which one looks first to (all) siblings of the current node, and then siblings of a parent, etc. Structure (A) would predict a choice of [1] or [2], equally. Structures (B) and (C) would have a preference for [2] as the antecedent (with (C) allowing [1] as a dispreferred option, and (B) disallowing it), while (D) would see the unit of [1] and [2] combined as the most likely antecedent, i.e., not necessarily a rejection of [2] in and of itself, but of [1] and [2] together as the realization of the goal that inspired production of [1] (for reasons that might be due to problems with either of the utterances/actions themselves, or the coherence of the two).

3 Examples

In order to decide on which structure is most appropriate, as well as what kinds of representations are needed for the task, it will be helpful to examine a suite of instantiations of the exchange schema in (1). We first examine some examples of corrections to exchanges found in human-human task

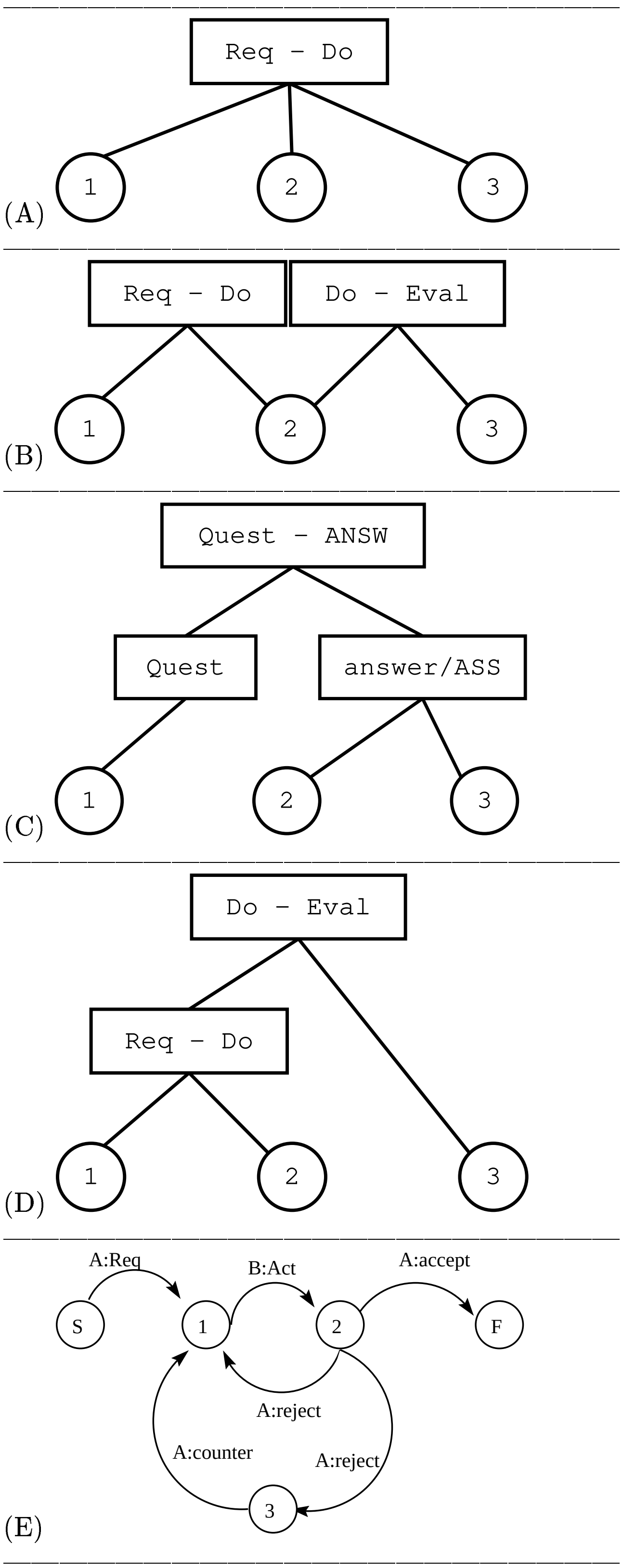

Figure 1: Proposed Structures for IRF unit

oriented dialogue, specifically the Maptask. We then turn to the TRAINS-96 domain, abstracting some simpler examples that can straightforwardly be tested in a spoken dialogue system.

3.1 Maptask Corpus Examples

We draw examples of correction exchanges from the DCIEM (examples 3 - 7) [33] and HCRC [5] (example 8) Maptask corpora. In the dialogues comprising these corpora, two people are looking at their own versions of maps of the same territory. The maps are slightly different, and neither can see the other person's map. One person (the Giver, indicated by [G]) is attempting to give directions to the other (the follower, indicated by [F]). For each excerpt, identifying information about the specific dialogue is given above.

In (3) the giver has requested an action: "go west", which the follower confirms. But then the giver changes his mind (realizes he has made a mistake) and corrects to: "[go] east".

(3) +<text id=r120.f.q2c5.10.2-4.3-1>

```
[G]: You'll ...  You'll go north ...  and then
     you'll turn west, onto the bridge.
[F]: Okay.
[G]: Or, east.  Correction.
[F]: Okay.
```

In (4), the requested action, go west at the bottom of the ravine itself, has been (slightly) misunderstood as a request to go west on the word 'ravine' despite explicit instructions to the contrary.

(4) +<text id=r110.p.q4c3.36.5-6.11-3>+

```
[G]: then curve back out...and then, at
     the bottom of the ravine, not the word
     \{ci|ravine\}, the ravine itself...
[F]: \{gg|Uh-huh\}
[G]: you're going to head west.
[F]: Okay.  So basically I'm going to be on
     the...I'm going to curve...and I'm going
     to be on the west side of the ravine?
[G]: Right.
[F]: And then, on the word \{ci|ravine\}, go
     west, right?
[G]: No.  In line with the bottom of the
     ravine itself ...
[F]: Okay.
[G]: go west.
```

Example (5) is a little trickier, since the action request is implied: [look] in the area of the playground [and tell me what you see]. The giver clarifies the action: look to the east of that ([are you looking] 'to the east of it?'). The follower reports a tyre swing. 'No. To the east of it, further.' Here the giver has used an imprecise direction, [look] 'to the east', which has been disambiguated incorrectly by the follower (look only a *little* to the east). When the follower reports what he sees, the giver repeats the same request, 'to the east of it', but then clarifies with the more precise 'further'.

(5) +<text id=r120.f.q3c5.22.2-4.7-1>+

[G]: Okay ... Have you got anything in the \{fp|uh\}
adventure playground area? Like...
[F]: Yeah, they're ... tyres?
[G]: To the east of it?
[F]: A tyre swing?
[G]: \{fp|Uh\}, no, to the east of it, further. My map's
empty here.
[F]: A privately owned fields?
[G]: Privately owned fields.

In (6), the giver requests an action: 'curve toward the train crossing ...' containing an ambiguous designator 'the train crossing' which the follower disambiguates in a way not consonant with the giver's intention. In this exchange the sign that this is the case comes when the follower realizes that the rest of the instruction: 'up along the west side of the waterfall' is incompatible with 'towards the train crossing.' He initiates a repair by explaining the position of *his* train crossing. The giver acknowledges the repair: 'no no, not that train crossing' and here it does not seem far-fetched to claim that there is an implied: 'the other train crossing.' The interspersed confirmations (F: Okay. G: Okay ...) seems to bear out the supposition that there is a task (searching for the other train crossing) which is being carried out. In this case, the task fails, because the giver's map has no other train crossing, and other repairs must be attempted to allow a complete interpretation of the giver's original intention. In the current case the *sign* of the problem with the disambiguation is the contradiction between two parts of the giver's request, and this contradiction is noticed by the follower.

(6) +<text id=r120.f.q3c5.22.2-4.7-1>+

[G]: Okay, curve down towards the train
crossing ... and up along the west side
of the waterfall. Stop when you get to
the top of the waterfall.
[F]: Wait, my train crossing is \{fp|um\}
\{br|north west=northwest\} of the
waterfall.
[G]: \{fg|Oh\} no no, not that train crossing.
[F]: Okay.
[G]: Okay ...
[F]: I only have a waterfall, then.

Example (7) is more straightforward: the follower, in response to a complicated and ambiguous set of instructions (which are not worth taking space with here), checks his understanding by saying: 'I'll go right to the rope rope bridge'. The giver things this is a mistake, that the user is going to *turn* right to get to the bridge, rather than go straight there. So he gives a less ambiguous instruction: 'go due north' However, the follower had the correct intention all along.

(7) +<text id=r130.f.q2c5.10.2-4.3-1>+

[F]: I'll go right to the rope rope bridge.
[G]: No, you want to go due north ...
[F]: Yeah.

Example (8) is a common exchange for anyone giving real-time directions, e.g. while driving in a car. The request is 'go to the left.' The follower does the correct thing, but says 'right, OK' which is understood as being the *opposite* of what the giver wanted. Hence the re-assertion of the direction, in this case in just the same form: 'No, [go] left'

(8) (HCRC Maptask Corpus `[q2nc7.trn]`)

```
[G]: So, \{fg|eh\}, go to the left two inches.
[F]: \{fg|Eh\}, right, okay.
[G]: No, left.
```

3.2 Example Suite

For more easy comparison, we cast various types of repair, such as those found in examples (3 - 8), above, from the maptask corpus in a suite of minimally different examples. We use the TRAINS-96 domain [3], in which a user interacts with a dialogue system to provide routes for trains. Figure 2 illustrates an episode from this task, in which there are two trains of interest, Northstar, which is currently at Boston, and Metroliner, which started the task at Boston, but is now at Albany. Given this same context, consider the dialogues in (9) through (15).

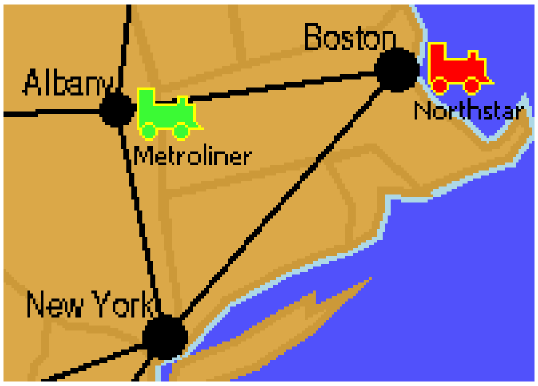

Figure 2: Trains Scenario

(9) [1] A: "send Northstar to New York."
[2] B: [sends Northstar to NY]
[3] A: "no, send Metroliner."

(10) [1] A: "send Metroliner to New York."
[2] B: [sends Northstar to NY]
[3] A: "no, send Metroliner."

(11) [1] A: "send the Boston train to New York."
[2] B: [sends Northstar to NY]
[3] A: "no, send Metroliner."

(12) [1] A: "send the Boston train to New York."
[2] B: [sends Northstar to NY]
[3] A: "no, send the Boston train."

(13) [1] A: "send the Boston train to New York."
[2] B: [sends Metroliner to NY]
[3] A: "no, send Metroliner."

(14) [1] A: "send Metroliner to New York."
[2] B: [sends Metroliner to NY]
[3] A: "no, send Metroliner."

(15) [1] A: "send Northstar to New York."
[2] B: [sends Metroliner to NY]
[3] A: "no, send Metroliner."

In each of these, the semantic structure of utterance [3] is something like (16). Context is used to determine what "X" refers to, and also to construe "Y" to be appropriately coherent, if possible.

(16) `Don'tDo(X) & Do(Y)`

Example (9) is similar to corpus example (3) — a changed instruction. Example (10) is a very common sort of exchange, where, as in example (4), the second contribution indicates a misunderstanding of the original. Examples (11) and (12) involve clarifications to an ambiguous reference. The tyre swing (5) and train crossing (6) examples are similar to these types. The successful adjudication of the problems in the exchange will rely on treating the combined user utterance as the expression of a single intention : 'send the Boston Train to New York ... no, send the Boston Train' which is only *incoherent* in the case that the system disambiguates 'the Boston Train' the same way each time. This strikes us as very similar to reasoning required in maptask example (6), in which the giver's instructions are only incoherent on one interpretation of the designator 'the train crossing.' The difference is only that there is no accessible option for the follower in (6) for an alternate disambiguation (there *is* no other train crossing on the follower's map), and this fact dictates a different repair strategy. Example (13) is very similar to the rope bridge example (7), although this may not be immediately apparent. The potential difficulty is that, although in the examples from the corpora the feedback from the follower is verbal, in the case of the TRAINS-96 system, the feedback is visual: one sees the train move. It is easy to imagine that the verbal report of the follower being misinterpreted, but less easy to imagine that the same ambiguity could arise in the case of the clear image of trains moving around the screen: one can simply *see* whether the intention is being carried out. But this is not necessarily so: the trains in the TRAINS system are color-coded, rather than labeled with their names.[1] Thus the user *sees* the green train moving (which is in fact the correct train for the system to move) but thinks it should be the red train moving. Assuming that the problem is the ambiguity of the initial instruction, the user is more specific, but the system has done the right thing all along. Example (14) is very similar to the left-right-left corpus example (8). Example (15) can be seen as a slight modification of (9): the user has changed his mind, however in this case, the system has complied with the new rather than originally expressed intention. The user may either be unaware of (or have misunderstood) the actual action, or want to avoid a possible repair by the system, upon recognizing the discrepancy between original directive and action.

[1] The name labels for the trains in Figure 2 were added for illustrative purposes, and not part of the actual display.

3.3 Examples and Structures

Let us now turn back to the different structures in Figure 1, and see if the examples from the previous sections have any implications for which structures would be most useful. The coherence of Dialogue (9) but lack of coherence of (15) indicates a problem with (A): it seems that after [2], [1] is no longer a possible antecedent in the same way. The contrast between (9) and (10) shows that the problem with [2] can be either a lack of coherence with [1], or a change in intention. This would seem to be a problem for (B), which does not preserve [1] as part of the context for [3]. Likewise, in (11) and (12), the source of the problem is likely the interpretation of the referring expression, "the Boston train". This is important for interpreting [3] coherently in (12), and recognizing (13) as incoherent. It is less easy to see how this information can be retrieved from (C) as opposed to (D). In general, for these examples, the source of the correction in [3] can be anywhere in the space including what A actually said in [1] (true 3rd turn repair), B's interpretation of that, or B's response in [2] (2nd turn repair). (D) seems to be the most useful candidate representation, since it provides the complex act of [1] and [2] together as a likely antecedent for [3]. (E) captures the move sequences correctly, but does not help much with the referential dependencies.

The interesting issue for these examples is how to respond to [3] in each case. For examples (9), (10) and (11), B can just undo the action performed in [2] and proceed to do the one mentioned in (3). In fact, this is just what the Rochester TRAINS-96 system [3] will do. For (12), the situation is a bit more complex. B must recognize that the previous choice of anchor for the referring expression "the Boston train" was likely to be wrong, and choose a different candidate, given A's response. For (13), (14) and (15), there is no obvious strategy to make [3] coherent, so some sort of repair would be warranted to overcome the incoherence. In order to be able to engage in fruitful dialogue rather than just respond to a sequence of commands, the important thing to realize is that [3] is a complex command with structure like (16), rather than unrelated `cancel` and `request` acts. To engage in natural dialogues, it is important to find a coherent interpretation when possible, and note and repair the incoherence when it is not possible to find such an interpretation.

4 Our Approach: Internal Representations

Our approach to the problem of representing local sub-dialogue structure of an IRF unit such as (2) is to represent not just the moves [1], [2], and [3], themselves, as part of the IRF unit, but also, like [20], some associated internal structures, which can help provide likely candidates for resolving any seeming incoherence. Thus, our counterpart of the Req-Do sub-structure in Figure 1 (D) includes not just the two acts, but each of the following components:

1. **L-req** (for "literal", or "locutionary") the actual words said.
2. **I-req** (for "interpreted", "intentional" or "illocutionary") the direct logical interpretation. This level maintains all ambiguity present in the original, including lack of a specific referent for "the Boston train".
3. **D-req** (for "disambiguated" or "domain") a precisification of the I-req that actually represents a specific request for an action that can be

performed by the domain module. For simple, unambiguous requests, in which the representation output by the language module and used by the domain module are the same, D-req can be just about identical with I-req; for cases with ambiguity or divergences in representation, it may involve several operations to get from I-req to D-req. D-Req represents *what* should be done in a manner that the domain reasoner can understand.

4. **P-act** (for "plan") a specification of *how* to do the requested act in D-req. A plan suitable for execution, which, if carried out will satisfy the original request
5. **E-Act** (for "execution") the action the system actually takes in fulfilling the request, which could be a physical (or simulated) action, or natural language production, or some combination.
6. **O-Act** (for "observation") concerns monitoring or observation of the system's act. Even if the system performed the act correctly, it might not have evidence of this fact. For linguistic actions, this is related to *grounding* [7, 34]. Grounding involves adding to the common ground between conversational participants, e.g., by presenting material and acknowledging and/or repairing presented material. For linguistic actions, or actions whose only evidence is through linguistic reports (e.g., the maptask examples in Section 3.1), the grounding process is the best indication of the success or failure of the action. Likewise, for environments such as TRAINS, in which there is visual evidence, monitoring the reactions to these acts are a primary method for grounding, as well as telling whether requested acts are successful (in terms of meeting the intentions of the requester).

As said above, the interpretation of D-req from I-req could involve several intermediate actions. In the case of dialogues (11), (12), and (13), it involves construction of a new query (corresponding to "which engine is the Boston engine"), calculating the answer to this query (perhaps using messages to a domain reasoner), and then fitting this answer into the D-req for the main request (replacing the more indirect information present in the I-req) I.e., in processing (13), "Boston train" will give "train at Boston" at the I-req level, but "Metroliner" in D-req.

In the example dialogues, utterance [1] has the same L-req and I-req in (11), (12), and (13) (though different from (9) and (14)). For (11) and (12), the D-req is the same as in (9), while in (13), the D-req is the same as (14), depending on the interpretation of "the Boston train" as Northstar or Metroliner, respectively.

Using this more fine-grained notion of the Req-Do unit, we can reexamine the likely sources for the correction in [3] in each of the cases. For (9), the obvious interpretation is that there was a problem with I-req: either A mis-spoke in [1], or changed his mind, or B misheard; pinpointing the exact source of the problem is not important, given that the same action can be taken to rectify the situation in each case. For (10), B must have misheard (or somehow made a mistake in execution). For (11), the most natural interpretation is that there was a problem at the D-req level, and A meant Metroliner rather than Northstar. For (12), things are a bit more subtle. Probably the problem is the same as for (11), but less information is provided by A about the correction — B must use the information that Northstar is probably not the correct choice when interpreting the repair. For (13), (14), or (15), the problem is most likely with P-act or E-act, or

L-req (i.e., in the speech recognizer, but then with L-req for [1] or [3]?) or some unresolvable contradiction. With luck, the confusion can be cleared up using a subdialogue with the user. While it is not always crucial to identify the exact source of the problem, it is important to recognize these situations of incoherence when they occur, and not just undo the previous act and redo the very same thing.

4.1 Repair at various levels

In addition to being able to repair when faced with an unresolvable contradiction, as in dialogues (13), (14) and (15), repair is also an option whenever there is difficulty computing any of these components of the representation, or when the system is insufficiently confident of its computation. Some examples of the kinds of repairs that concern difficulties at the different levels are shown in (17).

(17) **L-req:** "what was the third word?"

I-req: "is Metroliner an engine?"

D-req: "which train did you mean when you said 'Boston train'?"

P-act: "is going through Albany an appropriate way to send Metroliner to NY?"

E-Act: "should I do that now or after I send Bullet through?"

O-Act: "is it there now?"

A central issue for dialogue management becomes which strategy to use when facing uncertainty about the user's intent. E.g., in the case of two possible candidates for the referent of an expression like "the boston engine", one could either pick one and try it, or ask for clarification, as above. The decision should be motivated by factors such as how difficult it will be to correct a mistake and the likelyhood of picking incorrectly – in general users have litttle tolerance for multiple confirmations when things are going well. The representations here give a general set of possibilities from which to choose, based on actual circumstances, while current systems mainly allow only pre-designed decision points. See [35] and [14] for some ideas on how to use utility theory to choose which action to perform.

5 Implementation: ACDM

We have implemented this approach to dialogue representation in the Alma Carne Dialog Manager (ACDM), using Active Logic [10, 9, 13]. The dialog manager and reasoner are relatively domain and system independent, relying however on translation actions to convert between the internal logic and external system components. Alma (Active Logic MAchine) (described briefly in [25]), the current implementation of active logic, combines logical reasoning in time with an ability to perform and monitor the progress of external actions. Carne is used to model aspects of the agent's behavior which need not be expressed logically. Carne can run procedures for Alma, and acts as the I/O channel to Alma. ACDM is aimed at achieving a higher degree of *conversational adequacy* [22] than other current dialog systems.

5.1 Active logic

Active logics [10, 13, 23] were developed as a means of combining the best of two worlds – inference and reactivity – without giving up much of either. This requires a special evolving-during-inference model of time. The motivations for this were twofold: all physically realizable agents must deal with resource limitations the world presents, including time limitations; and people in particular have limited memories [4] and processing speeds, so that inference goes step by step rather than instantaneously as in many theoretical models of rationality. A consequence of such a resource-limited approach is that agents are not (even weakly) omniscient: there is no one moment at which an agent has acquired all logical consequences of its beliefs. This is not only a restriction for real agents (and hence for humans) but it is also an advantage when the agent has contradictory beliefs (as real agents will often have, if only because their information sources are too complex to check for consistency). In this case, an omniscient and logically complete reasoner is by definition swamped with all sentences of its language as beliefs, with no way to distinguish safe subsets to work with. By contrast, active logics, like human reasoners, have at any time only a finite belief set, and can reason about their present and past beliefs, forming new beliefs (and possibly giving up old ones) as they do so; and this occurs even when their beliefs may be inconsistent. (See [21] for details.)

Active logics can be seen either as formalisms per se, or as inference engines that implement formalisms. This dual-role aspect is not accidental: it is inherent to the conception of an active logic that it have a behavior, i.e., the notion of theoremhood depends directly on two things that are not part of traditional logics: (i) what is in the current evolving belief set, and (ii) what the current evolving time is. Our view of active logic here is as an on-board agent tool, not as an external specification for an agent.

5.1.1 Formalism

The formal changes to move from a first order logic to an active logic are, in some respects, quite modest. The principal change is that inference rules become time-sensitive. The most obvious case is that of reasoning about time itself, as in the rule:

```
i:             Now(i)
              ----------
i+1:           Now(i+1)
```

The above indicates that from the belief (at time i) that the current time is in fact i, one concludes that it *now* is the later time $i+1$. That is, time does not stand still as one reasons.

Note that temporal logics [2, 19, 26] also have a notion of past, present and future, but these do not change as theorems are derived. These are specification logics external to the reasoner. This contrasts strongly with the agent-based on board character of active logic.

Technically, an active logic consists of a first-order language, a set of time-sensitive inference rules, and an observation-function that specifies an environment in which the logic "runs". Thus an active logic is not pure formalism but is a hybrid of formal system and embedded inference engine, where the formal behavior is tied to the environment via the observations and the internal monitoring of time-passage (see [10] for a detailed description). Further formal details are given below.

5.1.2 Properties of active logic

Active logics are able to react to incoming information while reasoning is ongoing, blending new inputs into its inferences without having to start up a new theorem-proving effort. Thus, any helpful communications of a partner (or user) – whether as new initiatives, or in response to system requests – can be fully integrated with the system's evolving reasoning. Similarly, external observations of actions or events can be made during the reasoning process and also factored into that process.

Thus the notion of theorem for active logics is a bit different from that of more traditional logics, in several respects:

1. **Time sensitivity.** Theorems come and go; that is, a proposition once proved remains proved but only in the sense of it being a historical fact that it was once proved. That historical fact is recorded for potential use, but the proposition itself need not continue to be available for use in future inferences; it might not even be reprovable if the "axioms" (belief) set has changed sufficiently. As a trivial example, suppose $Now(noon) \rightarrow Lunchtime$ is an axiom. At time t=noon, Now(noon) will be inferred from the rule given earlier, and Lunchtime will be inferred a step later. But then Now(noon+1) is inferred, and Lunchtime is no longer inferable since its premise Now(noon) is no longer in the belief set. Lunchtime will remain in the belief set until it is no longer "inherited"; the rules for inheritance are themselves inference rules. One such involves contradiction; see next item.

2. **Contradictions.** If a direct contradiction (P and $\neg P$) occurs in the belief set at time t, that fact is noted at time t+1 by means of the inference rule

```
t:                   P , ~P
                 ------------------
t+1:              Contra(t+1,P, ~P)
```

   See [21] for details on handling contradictions

   Truth maintenance systems [8] also tolerate contradictions and resolve them typically using justification information. This happens in a separate process which runs while the reasoning engine is waiting. We do not think that this will work in general since the reasoning needed to resolve the contradiction will depend on the very information that generated that contradiction. Resolution of contradictions is itself, in general, a reasoning process much like any other.

3. **Metareasoning**

   In active logic, there is a single stream of reasoning, which can monitor itself by looking backwards at one moment to see what it has been doing in the past, including the very recent past.

   All of this is carried out in the same inferential process, without the need for level upon level of meta-reasoners. This is not to say that there is no metareasoning here, but rather that it is "in-line" metareasoning, all at one level. The advantages of this are (i) simplicity of design, (ii) no infinite regress, and (iii) no reasoning time at higher levels unaccounted for at lower levels. A potential disadvantage is the possibility of vicious self-reference. This matter is a topic of current investigation. However the contradiction handling capability should be a powerful tool even there.

5.2 Alma/Carne

Alma is our implementation of active logic. It generally conforms to the description of active logic given above, with some variations for greater efficiency or ease of implementation. Alma is written in Prolog and has a Java user interface. Although Alma cannot currently be run across the web, reasoning episodes (saved in history files) can be viewed with the interface on our web-site (http://www.cs.umd.edu/projects/active/demos).

At each step, the inference rules are applied to the formulas in the KB at that step and the resulting set of formulas is the KB for the next step. Application of the inference rules can result in formulas being added or removed from the KB. By default though, all formulas are inherited from one step to the next (an exception being *now*). Some features of Alma are:

- The current step number, T, is represented in the KB as $now(T)$ and changes as the program executes. One can reason about the current time by using $now(T)$ in the axioms.
- Three variants of the conditional are available in Alma for better control of the reasoning:

  if This acts like the familiar material conditional.

  fif If $fif(\phi, conclusion(\psi))$ and ϕ are in the KB, then ψ is asserted. This does not allow contraposition: if $\neg\psi$ is in the KB, we do not obtain $\neg\phi$. Another feature of fif is that the antecedent must be in the KB before the fif is used. If, for example, ϕ is $\alpha \wedge \beta$ and α is in the KB, we cannot derive $fif(\beta, conclusion(\psi))$. Only if both α and β are in the KB at the same time will ψ be derived. fif formulas can only be used in forward chaining proofs.

  bif This is used to mark conditionals for use exclusively in backward chaining proofs. It can be used rather than if or fif to avoid generating large amounts of true but uninteresting facts, while still allowing the ability to prove interesting information on demand.

- If there is a direct contradiction (e.g., $\phi, \neg\phi$) in the KB, the formulas are made unavailable for use in further inference and $distrusted(\psi)$ is asserted, where ψ stands for each of the contradicting formulas (e.g., ϕ and $\neg\phi$) and their consequences. The fact that there is a contradiction is asserted: $contra(N1, N2, T)$ where $N1$ and $N2$ are the names of the contradictands (e.g., ϕ and $\neg\phi$) and T is the time at which the contradiction was detected.

  Asserting $reinstate(N)$ for a formula N that has been in a contradiction results in a new formula similar to N being added to the database. This can be used to resolve contradictions. The choice of which formula to reinstate is not determined by Alma inference rules, however Alma can use user-specified axioms to reason about how to make this choice.

- Some computations may be more easily, conveniently or efficiently done through procedures rather than as logical inference. The reserved predicate *eval_bound* is used to execute Prolog programs in the Alma process. This form allows one to specify that some variables need to be bound before executing the program. $eval\_bound(p(X, Y), [X])$ will execute program $p(X, Y)$ only if X is bound.

- Programs that are more complex or requre longer asynchronous run-time are executed in Carne (see details below). Carne also allows Alma to interface with external processes enabling Alma to be embedded in larger systems, such as the TRAINS-96 system [3].

5.2.1 Carne

Carne is a process that communicates with Alma but runs independently. The main purpose of Carne is to run complex non-logical computations asynchronously from Alma steps. These can include input-output, interfacing with other running systems and long-running computations that take too long to run within a step.

Alma requests actions from Carne by asserting formulas of the form $call(X, Y, Z)$ in the KB. A formula of this type is the premise of an inference rule which causes Carne to execute program X. Y is used to pass relevant information to Carne and Z is an identifier that links the call to assertions about the status of the call. The status of the execution: $doing(X, Z)$, $done(X, Z)$ or $error(X, Z)$ is asserted in the Alma KB based on information provided by Carne. As the call is executed, $doing(X, Z)$ is first asserted. When the program has completed in Carne, $doing(X, Z)$ is replaced with $done(X, Z)$ in Alma. If the program fails, $error(X, Z)$ is asserted instead. This enables Alma to reason about actions it has requested.

Carne can add and delete formulas in the Alma KB. This is used to modify the KB as a result of computations and for external input to be added to the KB. Carne also has a KQML parser which facilitates connection of Alma/Carne to other systems.

5.3 Maryland version of TRAINS-96

ACDM is a dialogue manager built using ALMA and Carne. It is integrated within the TRAINS-96 system from the University of Rochester [3], replacing their discourse manager. The TRAINS-96 system consists of a set of heterogeneous modules communicating through a central hub using messages in KQML [11]. This architecture is well suited for swapping in different components to do the same or similar job and assessing the results. As well as the architecture itself, we have been using the parser, domain problem solver, and display modules, replacing the discourse manager component with our own dialogue manager and multi-modal generator. The functions of the modules in the Maryland version of the system are summarized in (18).

(18) **Speech Recognizer:** produces a word stream from spoken utterance (using Microsoft's Whisper Engine).

Parser: produces interpretation of sentence input, as shown in Figure 3 (source for I-req).

Problem Solver: answers queries for problem state, also does planning requests (helps produce P-act from D-req).

Display Manager: shows objects on screen.

Dialogue manager: uses Active Logic to maintain a logical representation of dialog state and act appropriately.

Output Manager: provides multimodal presentations of system output, including calls to display manager, printed text, and speech.

```
(TELL :CONTENT
 (SA-REQUEST :FOCUS :V11916 :OBJECTS
  ((:DESCRIPTION (:STATUS :NAME) (:VAR :V11868) (:CLASS :CITY)
    (:LEX :TORONTO) (:SORT :INDIVIDUAL))
   (:DESCRIPTION (:STATUS :DEFINITE) (:VAR :V11879)
     (:CLASS :TRAIN) (:SORT :INDIVIDUAL)
     (:CONSTRAINT (:ASSOC-WITH :V11879 :V11868)))
   (:DESCRIPTION (:STATUS :NAME) (:VAR :V11916) (:CLASS :CITY)
    (:LEX :MONTREAL) (:SORT :INDIVIDUAL)))
  :PATHS ((:PATH (:VAR :V11908)
                 (:CONSTRAINT (:TO :V11908 :V11916))))
  :DEFS NIL :SEMANTICS
  (:PROP (:VAR :V11855) (:CLASS :MOVE)
   (:CONSTRAINT
    (:AND (:LSUBJ :V11855 :*YOU*) (:LOBJ :V11855 :V11879)
     (:LCOMP :V11855 :V11908))))
  :NOISE NIL :SOCIAL-CONTEXT NIL :RELIABILITY 100 :MODE KEYBOARD
  :SYNTAX ((:SUBJECT . :*YOU*) (:OBJECT . :V11879)) :SETTING NIL
  :INPUT  (SEND THE TORONTO TRAIN TO MONTREAL))
 :RE 3)
```

Figure 3: Parser Output for "Send the Toronto train to Montreal"

Speech Output: converts text messages to output speech (using the Festival system).

For purposes of illustration, we will consider a scenario of the same type as that shown in (12). In this scenario there are 3 trains in the domain: Metroliner, Bullet and Northstar, and Metroliner and Bullet are at Toronto. The initial user utterance will be : *"Send the Toronto train to Montreal"*. The output from the parser for this utterance (see Figure 3) includes the utterance type (sa-request), the objects mentioned in the utterance (1 train and 2 cities), and the properties of the objects (name, type etc.). The parser parses *Toronto train* as the train that is **associated with** Toronto.

5.4 KQML message processing

ACDM receives the message from the parser in KQML format which, when translated by Carne, causes formulas that represent the information content of the message to be asserted as axioms in the Alma database. For the current scenario, some of the assertions made to represent the information content are shown below. The number before the colon is an identifier for the formula that appears after the colon. The formulas represent information including the message number(kqml294), sender (parser), message type (sa-request) etc.

```
2732: kqml_expr(kqml296, [kqml297, kqml303, ...]))
2761: kqml_kv(kqml294, [content, kqml295])
2763: kqml_kv(kqml294, [sender, parser])
2765: kqml_head(kqml295, sa-request)
2767: kqml_kv(kqml295, [objects, kqml296])
```

```
2780: kqml_kv(kqml297, [var, v11868])
2781: kqml_kv(kqml297, [class, city])
2782: kqml_kv(kqml297, [lex, toronto])
2806: new_message_kv(kqml294)
...
```

5.5 L-req level

When it receives a message from the parser, Alma determines that the message content is a user utterance and hence asserts this fact in the database as shown below (2808). In addition, it realizes that the syntax of the utterance exists in its database as *kqml_kv* and *kqml_head* assertions.[2] Alma updates the list of utterances that it maintains to include the new message as an utterance. As a result, kqml294 gets added to the list of utterances as shown below(2814).

```
2808: lreq(utterance(kqml294), kqml294)
2814: utt_list([kqml294, kqml200, kqml98, ...]))
```

Once L-req level processing is completed, Alma reasons that the I-req level processing has to be carried out. The specific axiom that causes this reasoning is shown in (19). This uses the *fif* construction, which means that a new compute_ireq formula will be triggered for each new utterance. Having this formula in the database will also (under normal circumstances) trigger a call to carne to produce the ireq representation.

(19) fif(lreq(utterance(ID),ID),
conclusion(compute_ireq(ID))).

5.6 I-req level

At the I-req level, Alma requests from Carne an initial interpretation of the utterance. Carne uses the parser output represented in logic as *kqml_kv* and *kqml_head* assertions, to produce the initial interpretation. Some of the assertions that Carne makes in the Alma database during the processing at this level are listed below.

```
2823: ireq(type(kqml294, sa-request), kqml294)
2825: ireq(obj(kqml294, v11868), kqml294)
2830: ireq(lex(v11868, toronto), kqml294)
2832: ireq(class(v11868, city), kqml294)
2834: ireq(obj(kqml294, v11879), kqml294)
2837: ireq(at-loc(v11879, v11868), kqml294)
2841: ireq(class(v11879, train), kqml294)
2843: ireq(obj(kqml294, v11916), kqml294)
2848: ireq(lex(v11916, montreal), kqml294)
2850: ireq(class(v11916, city), kqml294)
2852: ireq(path(kqml294, v11908), kqml294)
2853: ireq(to(v11908, v11916), kqml294)
2855: ireq(sem(kqml294, v11855), kqml294)
2860: ireq(lf(v11855, [move, v11879, v11916]), kqml294)
2861: ireq(lex(v11879, null), kqml294)
2867: done(compute_ireq(kqml294))
```

[2]These are syntactic operators representing frame-type objects with a head and multiple keyword-value pairs.

Some examples of the kind of information in this interpretation include:

- identifying the speech act type of the utterance (eg., 2823)
- identifying the different objects (eg., 2843) and paths (eg., 2852) mentioned in the utterance
- associating the properties mentioned in the utterance to their corresponding objects/paths (eg., 2830, 2832, 2853)
- identifying the semantics of the utterance (eg., 2855, 2860)

Note in particular assertion (2837). Currently, the system automatically translates **assoc-with** (in the parser message) into **at-loc** (at location) when moving from the parser message to the I-Req level. This is a temporary shortcut in the implementation – once we have enhanced Alma to reason about other interpretations of 'associated with', Carne will be modified to pass back assoc-with at the I-Req level, to be disambiguated at the D-Req level.

In the present scenario, the semantics of the utterance represented using 2855 and 2860 is *move* $v11879$ to $v11916$ where $v11916$ is known to be *montreal* from 2848. However, $v11879$ has a lex value of *null*. All objects that have a *null* value for *lex*, are considered underspecified, since the referents for such objects are not directly available from the parser output. In the scenario under consideration, $v11879$ refers to the object designated by the user as *Toronto train*. In order to find the referent for such objects, additional background information is required (e.g., from the knowledge base in the domain problem solver) .

Once the initial interpretation is done, 2867 gets asserted and hence Alma reasons that D-req level processing has to be done to disambiguate all ambiguous objects. The axiom that triggers this reasoning is shown in (20).

(20) fif(done(compute_ireq(ID)),
conclusion(compute_dreq(ID))).

5.7 D-req level

At this level, ambiguous objects get disambiguated and the user intention is determined. In the present scenario, we would like object $v11879$ to be disambiguated as either *metroliner* or *bullet* as appropriate since those are the two trains at Toronto. We would also wish the user intention to be determined as: "move the object $v11879$ to Montreal".

The first step in the disambiguation process requires binding all intensional objects in IREQ to internally known object names. This is easy for names, since we assume a unique, fixed binding. For more complex referring expressions, this involves getting enough domain information to disambiguate the objects. In the current scenario, Alma requests that Carne find the current position of the trains in the system to determine the candidate set for the unbound object variable ($v11879$). Carne responds by asserting this set in the Alma database as follows:

```
2985: dreq(candidates(v11879, [metroliner,  bullet]), kqml294)
```

The choice of candidate is determined by coherence with previous information about user intentions. In the simple case, where no such information exists, ACDM simply picks the first item, and asserts that as the identification of the variable as shown below (2994). This now leads to an ascription of the disambiguated semantics of the request as the intention of the user, represented in 2997. We will return to the more general issue of reference resolution and user intentions in section 5.13 below, when considering the correction case.

```
2994: dreq(lex(v11879, [metroliner]), kqml294)
2997: move([metroliner], montreal)
```

When the D-req has been fully specified, Alma proceeds to P-act processing using the rule in (21).

(21) fif(done(compute_dreq(ID)),
conclusion(compute_pact(ID))).

5.8 P-act level

The response strategy is determined at this level. If the user intention involves changing the state of the world, then Alma requests Carne to obtain a plan to cause the required change of state. Carne consults with the problem solver and asserts the plan in the Alma database as a series of assertions as shown below. Carne also asserts the state in which the problem solver will be, if the plan were to be implemented (3205).

```
3176: pact(plan(plan679), kqml294)
3200: pact(action(kqml294, plan679, go709), kqml294)
3201: pact(type(go709, go), kqml294)
3202: pact(from(go709, toronto), kqml294)
3203: pact(to(go709, montreal), kqml294)
3204: pact(track(go709, montreal-toronto), kqml294)
3205: pact(psstate(kqml294, pss879), kqml294)
3207: done(compute_pact(kqml294))
```

The completion of the P-act level processing would cause the assertion of 3207, which in turn would trigger the axiom in (22).

(22) fif(done(compute_pact(ID)),
conclusion(compute_eact(ID))).

5.9 E-act level

Changing the problem solver (domain) state by executing the plan specified at the previous level and providing a response to the user are done at this level. Alma instructs carne to send messages to both the domain problem solver, to execute the above plan, and also to send a message to the output manager, to communicate this change of state to the user. The output manager currently has three choices of modality in which to express this information:

- NL Speech (via a call to the speech output module)
- NL Text (by displaying the text on the display window)
- Graphical Display (via a call to the Display manager to move or highlight trains, or draw or highlight paths)

Currently, for an execution like moving a train, the output manager chooses to use all three modalities, simultaneously speaking and displaying the message indicating the path to be used, while drawing and highlighting the path on the map.

Other kinds of output include coordinated use of modalities (such as highlighting a train or city to indicate reference), or use of only verbal modalities, such as for a clarification request.

Once Alma requests Carne to provide the natural language response and change the problem solver state if required, E-act level processing is complete. Hence, the following formula (3275) is asserted.

```
3275: done(compute_eact(kqml294))
```

5.10 O-act level

This level keeps track of domain changes; particularly it deals with confirming that the actions that the system initiated to cause a change in the domain state have been performed correctly. For instance, if a plan is successfully executed, then Alma makes a note of that fact.

In this scenario, the system is waiting for positive feedback from the user to deduce successful execution. In our example, instead of receiving such confirmation, the user provides a correction, "No, Send the Toronto Train to Montreal."

5.11 Handling rejection utterances

The parser output for the "no" utterance can be seen in Figure 4.

```
(TELL :CONTENT
 (SA-REJECT :FOCUS NIL :OBJECTS NIL :PATHS NIL :DEFS NIL
   :SEMANTICS :NO :NOISE NIL :SOCIAL-CONTEXT NIL
   :RELIABILITY 100 :MODE KEYBOARD
   :SYNTAX ((:SUBJECT) (:OBJECT)) :SETTING NIL :INPUT (NO))
 :RE 4)
```

Figure 4: Parser Output for "No"

This rejection goes through the previously mentioned levels of processing, and when it reaches the D-req level, the details of the previous utterance processing are examined to determine the intention behind the current utterance. (Although we plan to implement a more intelligent version of context sensitivity in the future, in our current implementation, we assume that the rejection implied in the statement *"No."* applies to the immediately preceding user utterance or system response.) If the preceding utterance was a request type utterance then we represent the intention behind the current utterance as the negation of the intention that has been ascribed to the previous utterance. In the current case, this causes the following assertion to be made in the Alma database, which in turn causes a contradiction in the system's beliefs.

```
3448: not(move([metroliner], montreal))
```

```
fif(and(contra(X,Y,Z),
    and(eval_bound(name_to_time(X,T1), [X]),
    and(eval_bound(name_to_time(Y,T2), [Y]),
        eval_bound(( T2 > T1 -> ReForm = X; ReForm=Y), [T1, T2,X,Y])))),
conclusion(reinstate(ReForm))).
```

Figure 5: Recency-based Contradiction handling axiom

The *contradiction detection inference rule* will now be applied, causing both 2997 and 3448 to be distrusted and 3450 to be asserted. Here 1494 denotes the time at which the contradiction was detected and the other numbers are all formula identifiers.

```
3450: contra(3448, 2997, 1494)
3451: distrusted(2997, 1494)
3452: distrusted(3448, 1494)
```

Assertion 3450 triggers the rule shown in Figure 5 which causes the most recently added of the contradicting formulas to be reinstated resulting in the following new assertion.[3]

```
3455: not(move([metroliner], montreal))
```

Thus the system, will now believe that "metroliner" should **not** be sent to Montreal and this information will be used in interpreting the next utterances especially in cases of reference resolution.

5.12 Interpreting the correction

Now the user repeats "Send the Toronto train to Montreal". Initial processing is the same as for the first utterance. When calculating D-req, when disambiguating "the Toronto train" we again have two choices: Bullet and Metroliner. But since Metroliner has been proscribed from moving to Montreal in the cancellation above, the disambiguation procedure picks Bullet:

```
3934: dreq(lex(v11992, [bullet]), kqml416)
```

This then results in the disambiguated user intention: Send Bullet to Montreal. The same steps are carried out for P-act and E-act. However, we assume that this time the user gives positive feedback, e.g., by going on to another instruction, or acknowledging, and we get at the O-act level:

```
3287: oact(plan_confirmed(kqml294, kqml390), kqml294)
```

[3]This is just one of many contradiction resolution strategies we are considering. This one is appropriate in the case of new information that is assumed to take precedence over prior information. In other cases, however, one would prefer more entrenched information, or other, perhaps perceptually-guided resolution strategies. Active logic and the Alma implementation give the expressive power needed to reason about these possibilities and consider which is best for which purposes).

But what if the user rejects that too and again repeats the original request? Given the state of the system, there will be no unrejected possibilities left. In this case, ACDM initiates a clarification subdialog with the user, by issuing the request: "Please specify the train by name" to find out exactly which train is to be sent. A specific train name will over-ride the previous intention information.

5.13 Intention-based Reference Resolution

Now let us re-consider the more general case of reference resolution, especially with consideration of the intentional constraints in (23) and (24).

(23) not(move([bullet],Montreal))

(24) not(move([metroliner],Montreal))

There are three relevant cases for using this information in guiding reference resolution in a case like formula 2985, with two plausible candidates. We take these in turn:

Without (23) and (24) in Alma database If neither of constraints (23) and (24) are present in the Alma database, both "Bullet" and "Metroliner" are equally likely candidates to resolve the contradiction. Therefore, as in the original sentence, ACDM chooses the first item in the list.

With (24) in Alma database If we only know (24), as in the correction case, above, the only likely possibility to resolve the reference for object v11879 is *bullet* and hence the system asserts the following intention in the Alma database.

```
3937: move([bullet], montreal)
```

With both (23) and (24) in Alma database If both (23) and (24) are present in the database, none of the candidates can be used to resolve the reference for object v11879. Hence, it would be advisable to get help from the user, by asking a clarifying question about the user's intention. In order to denote that the system has not been able to resolve the reference, ACDM asserts 5082 in its database. This will lead to a clarification request, as mentioned above.

```
5082: ref_confusion(kqml294, v11879, [bullet, metroliner]))
```

5.14 Evaluation

We have yet to undergo detailed user evaluations with the Maryland Trains system. However informal tests, such as the one described in this section, indicate important improvements over a purely object-level dialogue system, such as the original Rochester TRAINS-96 system. Given repetitions of "send the Toronto Train to Montreal", ... "No,...", the Rochester system will keep sending Metroliner (or whichever engine its separate reference resolution component selects) to Montreal. It never realizes that there is a miscommunication and therefore can't correct it, seeing each cancellation and action directive as sequential changes of the user's plan, regardless of the lack of overall coherence. ACDM, on the other hand. as described above, will recognize the contradiction and use this information to act differently on subsequent intepretation: clarifying if necessary, or simply performing a revised resolution if possible. This difference is a result of the deliberation ACDM does about its own reasoning and in particular about its previous conclusions regarding the user's intentions, which are understood to be relevant to understanding the user's current requests.

6 Related work and Discussion

There have been a number of multi-level approaches to dialogue. Grosz and Sidner presented a three-level structure composed of *Linguistic Structure*, in which utterances are conjoined into segments, *Intentional Structure*, in which the dominance and satisfaction-precedence relations of discourse segment purposes are related, and *attentional state*, in which focus spaces were stacked, for use in reference resolution [12]. Litman and Allen looked at discourse-level and domain-level plans, seeing the former as a class of meta-plans [16, 17]. Lambert and Carberry generalized this to also include a layer of problem solving plans [15]. Luperfoy looked at three types of structure relevant for reference: linguistic structure (the words mentioned), discourse pegs (the entities of the conversation, independent of the words used to refer to them, or the "real" objects they refer to), and belief (the actual entities) [18], with different rates of decay for the ability to refer to previous entities in these structures.

While these works were influential, they do not directly address the issues we have been concerned with here, of reasoning about the internal aspects of sub-dialogues, and in particular, supporting reasoning about contradictions.

Other systems have computed something like the six levels presented here, as part of their process of engaging in dialogue (e.g., the Underspecified Logical form and CRT representations in the TRAINS-93 system correspond to some degree to I-req and D-req [24, 37]). Where we are different from most researchers is in claiming the utility of keeping these levels as distinct representations for use as context in processing further utterances. Something like this is clearly necessary to deal appropriately with the examples we presented in Section 3. The Rochester system would do the same thing in each case: undo the previous action and interpret the second request in the restored context before the original request was fulfilled, with whatever train it decided upon for "the Boston Train" in (12). The ability to use the incoherence as a resource for recomputing a referential anchor or repairing is not available, nor is there an option of complaining about the seeming incoherence itself.

Keeping the I-level and D-level distinct is also important for sending appropriate messages back to the user. The I-level should be close to the linguistic structure of the user interaction, while the D-level should be close to what domain reasoners actually use. Conflating the two can lead to an inability to provide comprehensible feedback to the user. For example, the MIT Galaxy system [28] has several domain specialists, each used for a different kind of task. These domain reasoners use different ontologies, and thus, in their discourse representation (essentially the D-level), "Boston" is ambiguous between a `TOWN` in the CityGuide domain and a `CITY` in the AirTravel domain. The system may not be able to resolve which ontology object is being referred to, but surely a user not intimately familiar with the system internals would be very confused by a disambiguating query such as, "Do you mean Boston the city, or Boston the town". Fleshing this out with descriptions of the ontology types, such as "Boston the geographical area or Boston the point location" is not likely to help. Here, at the ontology of natural conversation (I-level), "Boston" is unambiguously the kind of entity that one could fly to or from, and which can contain restaurants, so any query would have to attack a different avenue for disambiguation, relating to the activities such as restaurant finding or flight booking, rather than to the kind of entity.

The approach that we are closest to, is perhaps [20], who also exploit the utility of maintaining multiple levels of representation as context. While there are some differences in the particular levels and type of structure assumed, a larger difference in approach is the uniformity of the representation language. McRoy, Haller, and Ali use a uniform approach, representing all aspects of processing in the same representation language, SNePS [30]. This does allow uniform reasoning and very powerful access to all parts of the representation, but also places limits on the kinds of language and domain subsystems that can be easily added to the system. Our approach is rather to treat the internals of the other subsystems more or less as black-boxes, interpreting only the final products within the logic.

Acknowledgments

This work was supported in part by AFOSR and ONR.The first author was also supported during some of the writing of this article by the Department of the Army under contract number DAAD 19-99-D-0046. Any opinions, findings and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the Department of the Army. We would also like to thank James Allen and George Ferguson from University of Rochester for allowing the use of the TRAINS-96 system as a platform in which to embed these ideas. Also, we would like to thank Ian Lewin, Ingrid Zukermann, and members of the TRINDI consortium and the Linguistics department of University of Gothenburg for helpful comments on previous versions of this material.

References

[1] Lars Ahrenberg, Nils Dahlbäck, and Arne Jönsson. Discourse representation and discourse management for a natural language dialogue system. In *Proceedings of the Second Nordic Conference on Text Comprehension in Man and Machine*, 1990.

[2] James F. Allen and George Ferguson. Actions and events in interval temporal logic. *Journal of Logic and Computation*, 4(5), 1994.

[3] James F. Allen, Bradford W. Miller, Eric K. Ringger, and Teresa Sikorski. A robust system for natural spoken dialogue. In *Proceedings of the 1996 Annual Meeting of the Association for Computational Linguistics (ACL-96)*, pages 62–70, 1996.

[4] A. D. Baddeley. *Human memory: theory and practice.* Allyn & Bacon, 1990.

[5] Jean Carletta, Amy Isard, Stephen Isard, Jacqueline C. Kowtko, Gwynewth Doherty-Sneddon, and Anne H. Anderson. The reliability of a dialogue structure coding scheme. *Computational Linguistics*, 23(1):13–31, 1997.

[6] Herbert H. Clark. *Arenas of Language Use.* University of Chicago Press, 1992.

[7] Herbert H. Clark and Edward F. Schaefer. Contributing to discourse. *Cognitive Science*, 13:259–294, 1989. Also appears as Chapter 5 in [6].

[8] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12(3):231–272, 1979.

[9] J. Elgot-Drapkin, S. Kraus, M. Miller, M. Nirkhe, and D. Perlis. Active logics: A unified formal approach to episodic reasoning. Technical report, Computer Science Department, University of Maryland, 1996.

[10] J. Elgot-Drapkin and D. Perlis. Reasoning situated in time I: Basic concepts. *Journal of Experimental and Theoretical Artificial Intelligence*, 2(1):75–98, 1990.

[11] External Interfaces Working Group. Draft specification of the kqml agent-communication language. available through the WWW at: http://www.cs.umbc.edu/kqml/papers/, 1993.

[12] Barbara J. Grosz and Candace L. Sidner. Attention, intention, and the structure of discourse. *Computational Linguistics*, 12(3):175–204, 1986.

[13] J. Gurney, D. Perlis, and K. Purang. Interpreting presuppositions using active logic: From contexts to utterances. *Computational Intelligence*, 13(3):391–413, 1997.

[14] Erik Horvitz and Tim Paek. Deeplistener: Harnessing expected utility to guide clarification dialog in spoken language systems. In *Proceedings 4th International Conference on Spoken Language Processing (ICSLP-00)*, Beijing, November 2000.

[15] Lynn Lambert and Sandra Carberry. A triparite plan-based model of discourse. In *Proceedings of the 29^{th} Annual Meeting of the Association for Computational Linguistics*, pages 47–544, 1991.

[16] D. J. Litman and J. F. Allen. A plan recognition model for subdialogues in conversation. *Cognitive Science*, 11:163–200, 1987.

[17] Diane J. Litman and James F. Allen. Discourse processing and common sense plans. In P. R. Cohen, J. Morgan, and M. E. Pollack, editors, *Intentions in Communication.* MIT Press, 1990.

[18] Susann LuperFoy. *Discourse Pegs: A computational Analysis of Content-Dependent Referring Expressions.* PhD thesis, University of Texas at Austin, 1991.

[19] D. McDermott. A temporal logic for reasoning about processes and plans. *Cognitive Science*, 6:101–155, 1982.

[20] Susan W. McRoy, Susan Haller, and Syed Ali. Uniform knowledge representation for language processing in the b2 system. *Journal of Natural Language Engineering*, 3(2/3):123–145, 1997.

[21] M. Miller. *A View of One's Past and Other Aspects of Reasoned Change in Belief.* PhD thesis, Department of Computer Science, University of Maryland, College Park, Maryland, 1993.

[22] D. Perlis, K. Purang, and C. Andersen. Convesational adequacy: mistakes are the essence. *Int. J. Human-Computer Studies*, 48:553–575, 1998.

[23] D. Perlis, K. Purang, D. Purushothaman, C. Andersen, and D. Traum. Modeling time and meta-reasoning in dialogue via active logic. In *Working notes of AAAI Fall Symposium on Psychological Models of Communication*, 1999.

[24] Massimo Poesio. *Discourse Interpretation and the Scope of Operators.* PhD thesis, University of Rochester, 1994. Also available as TR 518, Department of Computer Science, University of Rochester.

[25] K. Purang, D Purushothaman, D. Traum, C. Andersen, and D. Perlis. Practical reasoning and plan execution with active logic. In *IJCAI-99 Workshop on Practical Reasoning and Rationality*, 1999.

[26] N. Rescher and A. Urquhart. *Temporal Logic.* Springer-Verlag, New York, 1971.

[27] Emmanuel A. Schegloff and H. Sacks. Opening up closings. *Semiotica*, 7:289–327, 1973.

[28] S. Seneff, D. Goddeau, C. Pao, and J. Polifroni. Multimodal discourse modelling in a multi-user multi-domain environment. In Proceedings 4th International Conference on Spoken Language Processing (ICSLP-96), 1996.

[29] Kerstin Severinson Eklundh. The notion of language game – a natural unit of dialogue and discourse. Technical Report SIC 5, University of Linköping, Studies in Communication, 1983.

[30] Stuart C. Shapiro. The SNePS semantics network processing system. In Nicholas V. Findler, editor, *Associative Networks: Representation and Use of Knowledge by Computers*, pages 179–203. Academic Press, 1979.

[31] J. M. Sinclair and R. M. Coulthard. *Towards an analysis of Discourse: The English used by teachers and pupils.* Oxford University Press, 1975.

[32] S. Sutton, D. G. Novick, R. A. Cole, and M. Fanty. Building 10,000 spoken-dialogue systems. In Proceedings 4th International Conference on Spoken Language Processing (ICSLP-96), 1996.

[33] Paul A. Taylor, S. King, S. D. Isard, and H. Wright. Intonation and dialogue context as constraints for speech recognition. *Language and Speech*, 41:493–512, 1998.

[34] David R. Traum. *A Computational Theory of Grounding in Natural Language Conversation.* PhD thesis, Department of Computer Science, University of Rochester, 1994. Also available as TR 545, Department of Computer Science, University of Rochester.

[35] David R. Traum and Pierre Dillenbourg. Towards a normative model of grounding in collaboration. In *Proceedings of the* ESSLLI98 workshop on Mutual Knowledge, Common Ground and Public Information, 1998.

[36] David R. Traum and Elizabeth A. Hinkelman. Conversation acts in task-oriented spoken dialogue. *Computational Intelligence*, 8(3):575–599, 1992. Special Issue on Non-literal language.

[37] David R. Traum, L. K. Schubert, M. Poesio, N. G. Martin, M. Light, C. H. Hwang, P. Heeman, G. Ferguson, and J. F. Allen. Knowledge representation in the TRAINS-93 conversation system. *International Journal of Expert Systems*, 9(1):173–223, 1996.

[38] Gordon Wells, Margaret MacLure, and Martin Montgomery. Some strategies for sustaining conversation. In Paul Werth, editor, *Conversation and Discourse*. Croon Helm, 1981.