Int. Conf. on Intelligent Virtual Agents,
Santa Cruz, CA, Sept. 2012.

Using Collaborative Discourse Theory to Partially Automate Dialogue Tree Authoring

Charles Rich and Candace L. Sidner

Worcester Polytechnic Institute
Worcester, MA, USA
{rich,sidner}@wpi.edu

Abstract. We have developed a novel methodology combining hierarchical task networks with traditional dialogue trees that both partially automates dialogue authoring and improves the degree of dialogue structure reuse. The key to this methodology is a lightweight utterance semantics derived from collaborative discourse theory, making it a step towards dialogue generation based on cognitive models rather than manual authoring. We have implemented an open-source tool, called Disco for Games (D4g), to support the methodology and present a fully worked example of using this tool to generate a dialogue about baseball.

Keywords: cognitive models for behavior generation, conversational and story-telling agents

1 Introduction

Although the ultimate goal of most intelligent virtual agents research is to generate dialogue interaction from a rich underlying cognitive model, many current agents, for example in video games, are still developed using traditional dialogue tree authoring technology. Dialogue trees, however, have have many problems. We have developed a novel methodology that mitigates some of these problems by combining dialogue trees with hierarchical task networks. The benefits of our methodology include easier development and maintenance of large dialogues, greater reuse of dialogue authoring investment, and partial automation of dialogue generation. Most importantly, because our methodology is based on collaborative discourse theory, it is a step along the road toward more cognitively based dialogue interaction. The methodology is implemented in an open-source tool obtainable from the authors.

To quickly ground our discussion, Fig. 1 shows an example of the kind of dialogue interaction we are concerned with. This dialogue involving baseball games and news is taken from our current research project building a relational agent for isolated older adults.[1] Other dialogue interactions in the project involve

[1] See http://www.cs.wpi.edu/∼rich/always.

1 *Which team are you rooting for?*
2 • *Yankees.*
3 • *Red Sox.* ⇐
4 Really? But they aren't so great at winning the World Series!
5 • I bet you are a Yankees fan. ⇐
6 • Ah, but who cares? They play great ball!
7 No, I'm just joking with you.
8 • Oh. ⇐
9 • That's too bad, it would be more fun if you were!
10 *Did you catch Thursday's game?*
11 • *Yes.* ⇐
12 • *No.*
13 • *I don't want to talk about baseball anymore.*
14 What did you think of it?
15 • Awesome game! ⇐
16 • It was boring.
17 • We really needed that win.
18 Yeah, it was very intense. Great game.
19 • Yeah. ⇐
20 *I wonder how the Red Sox are doing in the standings. Should I check?*
21 • *Yes.*
22 • *No.* ⇐
23 • *I don't want to talk about baseball anymore.*
24 *Do you want to hear some recent baseball news?*
25 • *Yes.* ⇐
26 • *No.*
27 *Ok, I have several interesting stories here. The first one is about injuries. Would*
28 *you like to hear it?*
29 • *Yes.* ⇐
30 • *No.*
. . .
31 *Got time for another story?*
32 • *Yes.*
33 • *No.* ⇐
34 Well, that's about all. I sure like talking about baseball with you!
35 • Me, too. ⇐
36 • Let's talk again tomorrow.

Fig. 1. Example menu-based interaction (⇐ is user selection). *Italic lines* are automatically generated in the form shown in Fig. 2, with color added by the rules in Fig. 10.

1 *What is the Baseball favoriteTeam?*
10 *Shall we achieve LastGameDialogue?*
13 • *Let's stop achieving Baseball.*
20 *Shall we achieve BaseballBody by standings?*
24 *Shall we achieve BaseballBody by news?*
27 *Shall we achieve BaseballNews?*
31 *Shall we achieve BaseballNews again?*

Fig. 2. Default versions of indicated lines in Fig. 1 before color added by rules in Fig. 10.

diet and exercise counseling and calendar event scheduling. Because these are goal-directed interactions, chatbot technology was not appropriate.

Notice that Fig. 1 is a menu-based interaction, so that the system needs to generate both the agent's utterances and the user's menu choices.

1.1 The Problems with Dialogue Trees

The traditional dialogue tree approach to implementing such interactions entails manually authoring a tree of all possible agent utterances and user responses. For example, Fig. 3 shows the dialogue tree that is used to generate the subdialogue in lines 4–9 of Fig. 1. Notice that only some of the lines in the dialogue tree actually appear in Fig. 1, due to the user's menu choices. Now, imagine the much larger dialogue tree required to represent the interactions resulting from all possible user choices in Fig. 1.

The main advantage of such dialogue trees is that they give the author direct and complete control over exactly what is said during the interaction. Furthermore, with the addition of typical advanced features, such as conditionals, side effects, goto's and computed fields, such dialogue trees can be quite flexible in terms of implementing the desired control flow in a particular application.

The main disadvantage of dialogue trees is that they are very labor intensive, both for initial authoring and subsequent modification and reuse. Our methodology addresses this disadvantage by partially automating dialogue generation. To preview our results, we automatically generated the semantic content of the 21 italicized lines out of the total 36 lines in Fig. 1. (Of these 21 generated lines, the 7 lines in Fig. 2 required additional manual effort to add "color" as described in Section 6).

Furthermore, dialogue trees are an unsatisfying solution from the standpoint of artificial intelligence research. In comparison, our methodology explicitly models (some of) the goals of an interaction and the meanings of (some) utterances relative to those goals.

Our methodology arose out of two important observations about dialogues such as Fig. 1. First, most dialogues are hierarchically structured collaborations, even if they include only utterances and no actions. What this means is that the overall dialogue has some goal, e.g., discussing baseball, which is decomposed

4 Really? But they aren't so great at winning the World Series!
5 • I bet you are a Yankees fan.
7 No, I'm just joking with you.
8 • Oh.
9 • That's too bad, it would be more fun if you were!
Ok, from now on I'm a Yankees fan.
• Great!
6 • Ah, but who cares? They play great ball!

Fig. 3. Dialogue tree underlying lines 4–9 of Fig. 1, as indicated.

```
10 LASTGAME: Did you catch {...}'s game?
11   • Yes
       GOTO THINK
12   • No
       GOTO STANDINGS
13   • I don't want to talk about baseball anymore.
       GOTO ...
14 THINK: What did you think of it?
     ...
       GOTO STANDINGS
20 STANDINGS: I wonder how the {...} are doing in the standings. Should I check?
```

Fig. 4. Tags and goto's needed if a traditional dialogue tree were used to represent lines 10–20 of Fig. 1. The {...} indicate computed fields (see Section 6).

into subgoals (subdialogues), such as discussing the last game, checking the standings, and so on, recursively.

Second, we observed that many of the lines in a typical dialogue have to do with what you might call the "control flow" within this hierarchical structure. For example, the user's choice in lines 1–3 will control which of two introductory subdialogues they enter. Similarly, the user's choice in lines 10–13 will control whether or not they enter a subdialogue regarding the last game or whether they end the overall baseball dialogue entirely. Eventually, of course, the conversation gets down to (sub-...)subdialogues that consist entirely of application-specific content, such as lines 4–9.

In the traditional dialogue tree approach, both the hierarchical structure and the control flow is collapsed into the same representation together with the application-specific content. Collapsing this information together causes many problems. To start, this approach requires tags and goto's to express control flow branches and joins. For example, Fig. 4 shows the pattern of tags and goto's that would be needed in a traditional dialogue tree to represent the control flow in lines 10-20. This kind of "goto programming" is well-known to be error prone, especially when the logic is being frequently modified. Furthermore, such tangled webs of goto's make it difficult to reuse parts of the dialogue in other situations.

1.2 A Hybrid Methodology

Our solution to the problems with dialogue trees has been to evolve a hybrid methodology in which we use a hierarchical task network (HTN) to capture the high-level task (goal) structure and control flow of a large dialogue, with relatively small (sub-)dialogue trees attached at the fringe of the HTN. As we will see in detail below, making the hierarchical task structure of the dialogue explicit makes it possible to automatically generate much of the interaction shown in Fig. 1. The high-level task structure is also the part of the dialogue that most often gets reused. Furthermore, because all of the subdialogues, such

as Fig. 3, are at the fringe of the HTN, there is no need for goto's—all of the subdialogues "fall through" to the control structure of the HTN.[2]

To summarize our methodology:

- We start by laying out the hierarchical goal structure of the dialogue.
- Then we formalize the goal structure and control flow as an HTN.
- Next we add application-specific subdialogues at the fringe of the HTN.
- We iteratively test and debug the hybrid representation.
- Finally, we add color to the automatically generated utterances as desired (e.g., the difference between the lines in Fig. 2 and Fig. 1).

At the end of this process we often have a high-level goal structure that can be reused in other similar applications. For example, when we recently started building a basketball dialogue, we found that we could reuse the baseball HTN structure by substituting different subdialogues at the fringe. Bickmore, Schulman and Sidner [2] also experienced a high degree of reuse in applying a version of this methodology to health dialogues.

This methodology is supported by a tool, called Disco for Games (D4g), which is an extension of Disco, the open-source successor to Collagen [12, 13]. In the remainder of this paper, we explain in detail how each line in Fig. 1 is generated by D4g. First, Section 2 describes Disco's HTN formalism and how we have extended it in D4g to add dialogue trees at the fringe. Next, Section 3 describes Disco's lightweight utterance semantics, based on collaborative discourse theory, which is the key to the automatic generation of both agent utterances and user menus. Section 4 describes a small set of general rules that account for all of the automatically generated dialogue in the example (Section 5). Finally, Section 6 describes how application-specific formatting rules are used to add color.

2 Hierarchical Task Networks

Fig. 5 shows a diagrammatic summary of the HTN and dialogue tree hybrid structure underlying the interaction in Fig. 1. In this diagram, we follow the common simplifying convention of omitting nodes when a task has only a single recipe (decomposition) or a recipe (decomposition) has only a single step.

For the executable representation of HTN's we use the ANSI/CEA-2018 standard [10], on which Disco is based. Lines 1–17 of Fig. 6 show the XML syntax in ANSI/CEA-2018 for specifying the `Baseball` task, which is the toplevel goal of the dialogue, and its four steps (subgoals): `intro`, `lastGame`, `body` and `closing`. HTNs in this formalism include tasks with named and typed inputs, such as `favoriteTeam` (a `Team`) and `lastDay` (a `Day`) and outputs, and one or more recipes (`<subtasks>`)that decompose each non-primitive task into a sequence of (possibly optional or repeated) steps. Optionality and repetition are expressed together by the `minOccurs` and `maxOccurs` attributes of a step, both of which default to 1.

[2] Readers with a knowledge of the history of programming languages will recognize this as analogous to the argument for "structured programming" over goto's.

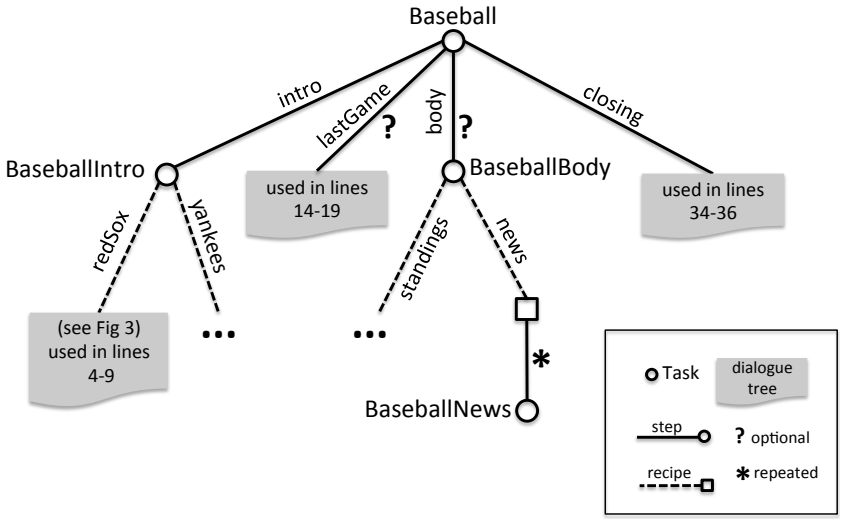

Fig. 5. Hierarchical task network underlying example interaction in Fig. 1.

```
<task id="Baseball">
  <input name="favoriteTeam" type="Team"/>
  <input name="lastDay" type="Day"/>
  <subtasks id="talk">
    <step name="intro" task="BaseballIntro"/>
    <step name="lastGame" task="LastGameDialogue" minOccurs="0"/>
    <step name="body" task="BaseballBody" minOccurs="0"/>
    <step name="closing" task="ClosingDialogue"/>
  </subtasks>
</task>

<task id="BaseballIntro">
  <subtasks id="redSox">
    <step name="intro" task="RedSoxIntroDialogue"/>
    <applicable> $Baseball.favoriteTeam==Team.ENUM.redSox </applicable>
  </subtasks>
  ...
</task>

<agent id="RedSoxIntroDialogue" text="Really? But they aren't...">
  <user text="I bet you are a Yankees fan.">
    <agent text="No, I'm just joking with you.">
      <user text="Oh."/>
      <user text="That's too bad, it would be more fun if you were!">
        <agent text="Ok, from now on I'm a Yankees fan.">
          <user text="Great!"/></agent></user></agent></user>
  <user text="Ah, but who cares? They play great ball!"/>
</agent>
```

Fig. 6. Part of ANSI/CEA-2018 and Disco for Games (D4g) specification of Fig. 5.

ANSI/CEA-2018 also supports the use of JavaScript to specify preconditions and postconditions of tasks and the applicability conditions of recipes. All of these conditions use a three-valued logic, where the JavaScript null value represents unknown. For example, the `<applicable>` element on line 14 selects the appropriate introductory subdialogue when the user's favorite team is the Red Sox.

Lines 18–26 of Fig. 6 are a straightforward XML encoding of the dialogue tree in Fig. 3, which is the leftmost dialogue tree on the fringe of the HTN in Fig. 5. The syntax used in these lines is transformed by D4g's XSLT preprocessor into appropriate ANSI/CEA-2018 specifications that cause the structure of the dialogue tree to unfold properly when executed in Disco. Thus, from a D4g author's point of view, both the HTN and the dialogue tree portions of the specification can be conveniently intermixed in a single file.[3]

3 Utterance Semantics

The key to automatically generating dialogue from the HTN portion of our hybrid representation is a lightweight semantics for dialogue utterances derived from Sidner's artificial language for negotiation [15] based on collaborative discourse theory [6, 8].

Collaborative discourse[4] theory is fundamentally an *interpretation* theory. It views dialogue as being governed by a hierarchy of tasks (goals) and a stack-like focus of attention and explains how to interpret an utterance (by either participant) as contributing to or changing the current task. Three fundamental ways that an utterance can contribute to a task are to:

1. provide a needed input (`Propose.What`)
2. select the task or a subtask to work on (`Propose.Should`), or
3. select a recipe to achieve the task (`Propose.How`).

In Disco, these three fundamental types of contribution are formalized, respectively, in the semantics of the first three builtin utterance types shown in Fig. 7 along with their default formatting. The semantics of these utterances also includes Sidner's model of the negotiation of mutual beliefs via proposal, acceptance and rejection [15]. Understanding these semantics is very important for the dialogue designer because, as we will see in the next section, they are the link between the HTN structure and the automatically generated dialogue utterances.

For example, when a dialogue participant utters a `Propose.What`, it means (in part) that the speaker:

- believes the proposition that the *input* to the *task* is *value* and
- intends that the hearer believe the same thing.

[3] D4g also supports transferring control to an HTN from inside a dialogue tree, so that HTN's and dialogue trees can in fact alternate in layers. However, we do not use this feature very often.

[4] In this work we consider only two-participant discourse, i.e., dialogue.

`Propose.What(`*task*`,` *input*`,` *value*`)`	*The task input is value.*
`Propose.Should(`*task*`)`	*Let's achieve task.*
`Propose.How(`*task*`,` *recipe*`)`	*Let's achieve task by recipe.*
`Accept(`*proposal*`)`	*Yes.*
`Reject(`*proposal*`)`	*No.*
`Ask.What(`*task*`,` *input*`)`	*What is the task input?*
`Ask.Should(`*task*`)`	*Shall we achieve task?*
`Ask.How(`*task, recipe*`)`	*Shall we achieve task by recipe?*
`Ask.How(`*task*`)`	*How shall we achieve task?*
`Propose.Stop(`*task*`)`	*Let's stop achieving task.*
`Ask.Should.Repeat(`*task*`)`	*Shall we achieve task again?*

Fig. 7. The main builtin Disco utterance types and their default formatting.

Similarly, uttering a `Propose.Should` proposes a task or subtask, such as an optional step, to work on. Uttering a `Propose.How` proposes a recipe to use. If the hearer `Accept`'s one of these proposals, then mutual belief in the respective proposition is achieved. The hearer can also `Reject` a proposal.

Utterances can also be questions. In Sidner's framework, questions are modeled as proposals by the speaker that the hearer provide information. For example, when a dialogue participant utters an `Ask.What` (see Fig. 7), it means (in part) that the speaker intends that the hearer respond by uttering a `Propose.What` that provides a *value* for the specified *task* and *input*. The other three utterance types starting with `Ask` have analogous semantics.

Disco implements these utterance semantics in its dialogue interpretation algorithm. Basically, to interpret a new utterance, the algorithm visits every live[5] task node in the HTN tree and asks the question, "Could this new utterance contribute to this task?" If the answer is yes, Disco attaches the new utterance as a child of the task node; otherwise it marks the utterance as "unexplained." (For more details about Disco's dialogue interpretation algorithm see [12].)

4 Generation Rules

Disco treats dialogue *generation* as the algorithmic inverse of interpretation. In other words, Disco visits every live task node in the current HTN tree and asks the question, "What are the possible utterances that *could* contribute to this task?" The answers to this question are the generation candidates.

Fig. 8 shows the overall functional flow of dialogue generation in Disco in more detail. Starting on the left, the first step is to apply the general generation rules described in this section to the current dialogue state, yielding a set of candidate utterances for either the agent or the user (depending on whose turn it is in the interaction). Each generation rule produces zero or more candidate utterances. These candidate utterances are then sorted according to heuristic priorities (see below). If utterances are being generated for the agent, then the

[5] A task is live if and only if its precondition is not false and all of its predecessor steps, recursively up the tree, have been successfully completed.

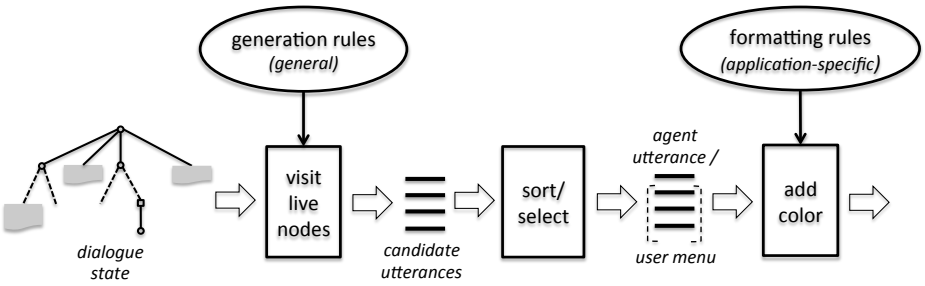

Fig. 8. Functional flow of dialogue generation in Disco.

```
define AskWhatRule (task)
  foreach input in inputs(task)
    if input does not have a value
      then return { new Ask.What(task, input) }
  return {}
```

Fig. 9. Pseudocode for `Ask.What` generation rule (applied to live task nodes).

highest priority candidate is chosen for the agent to speak; if utterances are being generated for the user menu, then the (perhaps truncated) candidate list is used to populate the user menu. Finally (see Section 6), optional application-specific formatting rules are applied to add color to some utterances, as desired.

Twelve general rules generate the content of all the lines in Fig. 1. These same rules are used in all of the dialogue applications we have built so far. There is one rule for each of the eleven utterance types in Fig. 7, plus an additional rule that generates the appropriate agent utterance or user menu entries from a dialogue tree on the fringe, when it is live.

Each generation rule is implemented as a small Java method that is applied by the generation algorithm to each live task node in the HTN tree as described above. For example, Fig. 9 shows pseudocode for the rule that generates `Ask.What` utterances. Notice that this rule will not return a question for a particular input if that input already has a value (which could happen either via dialogue or some other system process). In general, the generation rules only return candidate utterances when the relevant information is not already known.

Some rules behave differently depending on whether candidates are being generated for the agent or the user. For example, when the `Propose.What` rule is being executed for the user and the input type is an enumerated datatype, the rule returns a set of utterances (menu choices) that includes a `Propose.What` for each possible value.

Currently, because our agent is always subordinate to the user, the rules for `Accept` and `Reject` only generate candidates for the user menu. The inputs to these rules are task nodes that are dynamically added to the dialogue state whenever the agent makes a proposal (including questions). Notice that the

default formatting of `Accept` and `Reject` is simply "Yes" or "No," but that the underlying semantics includes the proposal being accepted or rejected.

The heuristic priorities used to sort candidate utterances have the most impact in agent generation, since only the topmost candidate is actually uttered by the agent. (In the case of the user, the priorities only affect the order in which the menu choices are displayed.) Currently, each of the twelve general rules has a fixed priority—we do not tweak priorities for a particular dialogue. Although these priorities are not yet grounded in any cognitive theory, they do follow a logical order of design decisions. For example, the `Ask.What` rule has a higher priority than the `Ask.How` rule, since the properties of the input to a task may affect the best recipe choice.

5 The Example Revisited

We can now revisit the example in Fig. 1 and explain how all of the italicized automatically generated lines are produced (at least in uncolored forms shown in Fig. 2). Starting with the agent utterances:

- Line 1 is generated by the application of the `Ask.What` rule to the `favoriteTeam` input of the `Baseball` task (see Fig. 6, line 2). Notice that no agent question is generated for the `lastDay` input of `Baseball`, because this input has already been bound as part of the system initialization.
- Line 10 is generated by the application of the `Ask.Should` rule to the `LastGameDialogue` step of `Baseball` (see Fig. 6, line 6).
- Line 13 is generated by the `Propose.Stop` rule, which only returns an utterance when applied to the toplevel goal of the dialogue (`Baseball` in this case). This rule provides the user with a convenient menu option for exiting the whole dialogue. This rule has some internal heuristics for when to return an utterance, depending on the dialogue state, so that the exit option is not always offered.[6]
- Lines 20 and 24 are generated by the application of the `Ask.How` rule to the `standings` and `news` recipes for `BaseballBody` (see Fig. 5).
- Line 27 is generated by the application of the `Ask.Should` rule to `BaseballNews` (see Fig. 5).
- Line 31 is an `Ask.Should.Repeat`, which is a variant of `Ask.Should` that is generated whenever the task being proposed is the second or subsequent instance of a repeated step, such as `BaseballNews`. The reason for this variant is to facilitate attaching a different formatting rule, as we will see in the next section.

Regarding the automatically generated user menu choices in Fig. 1:

- All of the Yes and No menu choices are generated by the `Accept` and `Reject` rules.

[6] Our agent currently automatically accepts all proposals by the user.

– Lines 2 and 3 are generated by the application of the `Propose.What` rule to the `favoriteTeam` input of the `Baseball` task. As mentioned above, this rule checks for the special case of enumerated datatypes, in which case it generates a `Propose.What` for each possible value. Furthermore, the default formatting for these utterances is simply the printable string for the data value. Enumerated datatypes in Disco are declared as JavaScript objects with an `ENUM` field (see Fig. 6, line 14).

In summary, we have now seen how all of the content in the example interaction is generated by the application of the general rules described above to the dialogue structure in Fig. 5. In the next section, we will see how the differences between the lines in Fig. 1 and in Fig. 2 are achieved.

6 Adding Color with Formatting Rules

The main reason why authors like dialogue trees is that it allows them to creatively tailor their use of language to the character and the narrative context—what we call "adding color" to the dialogue. In other words, the dialogues don't read like they were generated by a computer.

In the methodology we have evolved, we have found it best to postpone adding color until late in the authoring process. First, we develop and debug the HTN, such as Fig. 5, that represents the goal hierarchy and desired control flow between the fringe subdialogues. At the end of this phase, we have a working interaction that looks like Fig. 1, except with the corresponding lines from Fig. 2 appearing instead. Then we add color as desired via formatting rules. In this example, there were seven lines that needed color.

Formatting rules in Disco are specified in a Java properties file, which is organized as one key/value pair per line with an equal sign separating the key from the value, as shown in Fig. 10. Each key ends in `@format` with a prefix describing the type of utterance to which it is to be applied. For example, the rule in line 1 of Fig. 10 applies to all occurrences of `Ask.What` in which the *task* is `Baseball` and the *input* is `favoriteTeam`. The rule in line 10 applies to all occurrences of `Ask.Should` in which the *task* is `LastGameDialogue`, and so on. When a rule is applied, the value part of the rule is substituted for the default formatting of the corresponding utterance.

```
 1 Ask.What(Baseball,favoriteTeam)@format = Which team... rooting for?
10 Ask.Should(LastGameDialogue)@format = ...{$Baseball.lastDay}'s game?
13 Propose.Stop(Baseball)@format = I don't... about baseball anymore.
20 Ask.How(BaseballBody,standings)@format = I wonder... Should I check?
24 Ask.How(BaseballBody,news)@format = Do you want to hear... news?
27 Ask.Should(BaseballNews)@format = Ok, I have several interesting...
31 Ask.Should.Repeat(BaseballNews)@format = Got time for another story?
```

Fig. 10. Formatting rules applied to indicated lines in Fig. 2.

The rule on line 10 of Fig. 10 illustrates the use of computed fields, which is an important feature of the formatting system (that is also available for utterances that appear in dialogue tree). Curly brackets {...} in an utterance enclose arbitrary JavaScript code that is executed during the final formatting process to compute a string to substitute at that point in the utterance. In line 10, this feature is used to retrieve the value of the `lastDay` input of the most recently created instance of `Baseball`.

A commonly used special case of computed fields that is supported in Disco formatting rules (but not illustrated in this example) is using a vertical bar `|` to separate a set of alternative variations. For example, if `Accept@format` were set to `Ok|Yup|Sure`, the formatting system would systematically use these variations instead of all the Yes's in Fig. 1.

7 Related Work

Both HTNs [5] and dialogue trees [4] are very well known and commonly used techniques. HTNs have been used by others, such as Bohus and Rudnicky [3] or Smith, Cavazza et al. [16], for generating dialogue, but the goal of these efforts has been to eliminate dialogue trees, rather than coexist with them, as we have. Orkin et al. [9] have tried to eliminate manual dialogue authoring by applying data mining techniques to crowdsourced data to automatically create HTNs, which are then used to generate new dialogues.

This paper has grown out of our own previous work in several ways. In [11], we first described the idea of discourse generation as the algorithmic inverse of discourse interpretation and introduced the model of applying rules (called "plugins" in that paper) to the live nodes of an HTN to generate dialogue candidates, as shown in the first step of Fig. 8. We first described D4g in [7], although the emphasis in that work was on combining actions and utterances in a single representation, whereas this paper concerns itself entirely with utterances. The DTask system [1, 2], also an extension of ANSI/CEA-2018 and Disco, used HTNs with adjacency pairs (a single agent utterance with a user response menu) at the fringe, instead of complete dialogue trees, as in D4g. That work also explored the reuse advantages of HTNs in dialogue.

8 Conclusion

We have demonstrated, using an example baseball dialogue, how combining hierarchical task networks with dialogue trees greatly improves the authoring process as compared to using dialogue trees alone. Our methodology is supported by open-source tool, called Disco for Games (D4g), that is available by contacting the authors.

We have used D4g to author similar dialogues on other topics, such as family and weather, with similar positive experiences in terms of the number of automatically generated lines. (We do not quote the fraction of automatically

generated lines here as a statistic, because this number is easily skewed by the number of lines in the subdialogues at the fringe of the HTN.)

Looking toward the future, we see D4g as a step along the road toward totally automatic generation of dialogue. We expect to continue to extend the set of semantically specified utterance types (the current set is already in fact larger than shown in Fig. 7), which along with additional generation rules, will increase the amount of automatically generated content in dialogues. For example, we are interested in revisiting the automatic generation of tutorial dialogues, as we did in [14].

Acknowledgements. We would like to thank Fred Clinckemaillie and JJ Liu for their help with the baseball dialogue.

This material is based upon work supported by the National Science Foundation under Grant No. IIS-0811942 and IIS-1012083. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. Bickmore, T., Schulman, D., Shaw, G.: Dtask and Litebody: Open source, standards-based tools for building web-deployed embodied conversational agents. In: Proc. Intelligent Virtual Agents, Amsterdam, The Netherlands (2009)
2. Bickmore, T., Schulman, D., Sidner, C.: A reusable framework for health counseling dialogue systems based on a behavioral medicine ontology. J. Biomedical Informatics **44** (2011) 183–197
3. Bohus, D., Rudnicky, A.: The RavenClaw dialog management framework: Architecture and systems. Computer Speech and Language **23**(3) (2009) 332–361
4. Despain, W.: Writing for Video Games: From FPS to RPG. A. K. Peters (2008)
5. Erol, K., Hendler, J., Nau, D.: HTN planning: Complexity and expressivity. In: Proc. 12th National Conf. on Artificial Intelligence, Seattle, WA (July 1994)
6. Grosz, B.J., Sidner, C.L.: Plans for discourse. In Cohen, P.R., Morgan, J.L., Pollack, M.E., eds.: Intentions and Communication. MIT Press, Cambridge, MA (1990) 417–444
7. Hanson, P., Rich, C.: A non-modal approach to integrating dialogue and action. In: Proc. 6th AAAI Artificial Intelligence and Interactive Digital Entertainment Conf., Palo Alto, CA (October 2010)
8. Lochbaum, K.E.: A collaborative planning model of intentional structure. Computational Linguistics **24**(4) (December 1998) 525–572
9. Orkin, J., Smith, T., Roy, D.: Behavior compilation for ai in games. In: Proc. 6th AAAI Artificial Intelligence and Interactive Digital Entertainment Conf., Palo Alto, CA (October 2010) 162–167
10. Rich, C.: Building task-based user interfaces with ANSI/CEA-2018. IEEE Computer **42**(8) (August 2009) 20–27
11. Rich, C., Lesh, N., Rickel, J., Garland, A.: A plug-in architecture for generating collaborative agent responses. In: Proc. 1st Int. J. Conf. on Autonomous Agents and Multiagent Systems, Bologna, Italy (July 2002)

12. Rich, C., Sidner, C.: Collagen: A collaboration manager for software interface agents. User Modeling and User-Adapted Interaction **8**(3/4) (1998) 315–350 Reprinted in S. Haller, S. McRoy and A. Kobsa, editors, *Computational Models of Mixed-Initiative Interaction*, Kluwer Academic, Norwell, MA, 1999, pp. 149–184.
13. Rich, C., Sidner, C., Lesh, N.: Collagen: Applying collaborative discourse theory to human-computer interaction. AI Magazine **22**(4) (2001) 15–25
14. Rickel, J., Lesh, N., Rich, C., Sidner, C., Gertner, A.: Collaborative discourse theory as a foundation for tutorial dialogue. In: 6th Int. Conf. on Intelligent Tutoring Systems, Biarritz, France (June 2002) 542–551
15. Sidner, C.L.: An artificial discourse language for collaborative negotiation. In: Proc. 12th Nat. Conf. on Artificial Intelligence, Seattle, WA (Aug 1994) 814–819
16. Smith, C., Cavazza, M., Charlton, D., Zhang, L., Turunen, M., Hakulinen, J.: Integrating planning and dialogue in a lifestyle agent. In: Proc. Intelligent Virtual Agents, Tokyo, Japan (2008) 146–153