A Collaborative Problem-Solving Model of Dialogue

Nate Blaylock
Department of Computational Linguistics
Saarland University
Saarbrücken, Germany
blaylock@coli.uni-sb.de

James Allen
Deptartment of Computer Science
University of Rochester
Rochester, New York, USA
james@cs.rochester.edu

Abstract

We present a formal model of agent collaborative problem solving and use it to define a novel type of dialogue model. The model provides a rich structure for tracking dialogue state and supports a wide range of dialogue, including dialogue which contributes to interleaved planning and execution of domain goals.

1 Introduction

We are interested in building *conversational agents* — autonomous agents which can communicate with humans through natural language dialogue. In order to support dialogue with autonomous agents, we need to be able to model dialogue about the range of activities an agent may engage in, including such things as goal evaluation, goal selection, planning, execution, monitoring, replanning, and so forth.

Current models of dialogue are only able to support a small subset of these sorts of agent activities. Plan-based dialogue models, for example, typically model either planning dialogue (e.g., (Grosz and Sidner, 1990)) or execution dialogue (e.g., (Cohen et al., 1991)), but not both. Also, most plan-based dialogue models make the assumption that agents already have a high-level goal which they are pursuing.

In this work, we are trying to extend plan-based dialogue systems to more general *agent-based* dialogue systems (cf. (Allen et al., 2001)). In previous work (Allen et al., 2002; Blaylock et al., 2003), we presented a preliminary model of collaborative problem solving to model communicative intentions at the utterance level. In this paper, we formalize and extend that model and use it as part of a dialogue model that can represent dialogue about a larger range of agent activities — including those mentioned above. We also extend the collaborative problem-solving model to handle grounding phenomena by tying it together with a well-known model of grounding (Traum and Hinkelman, 1992).

The remainder of the paper is as follows: in Section 2, we give an intuitive definition of collaborative problem solving and then in Sections 3 to 6, we present the formalization of the collaborative problem solving model. In Section 7 we then extend the model to handle grounding. In Section 8 we show an example of the model on a dialogue. Section 9 then discusses related work, and then we conclude and mention future work in Section 10.

2 Collaborative Problem Solving

We see problem solving (PS) as the process by which a (single) agent chooses and pursues *objectives* (i.e., goals). Specifically, we model it as consisting of the following three general phases:

- *Determining Objectives*: In this phase, an agent manages objectives, deciding to which it is committed, which will drive its current behavior, etc.
- *Determining and Instantiating Recipes for Objectives*: In this phase, an agent determines and instantiates a recipe to use to work towards an objective. An agent may either choose a recipe from its recipe library, or it may choose to *create* a new recipe via planning.
- *Executing Recipes and Monitoring Success*: In this phase, an agent executes a recipe and monitors the execution to check for success.

There are several things to note about this general description. First, we do not impose any strict ordering on the phases above. For example, an agent may begin executing a partially-instantiated recipe and do more instantiation later as necessary. An agent may also adopt and pursue an objective in order to help it in deciding what recipe to use for another objective.

It is also important to note that our purpose here is not to specify a specific *problem-solving strategy* or prescriptive model of how an agent *should* perform problem solving. Instead, we want to provide a general descriptive model that enables agents with different PS strategies to still communicate.

Collaborative problem solving (CPS) follows a similar process to single-agent problem solving. Here two agents jointly choose and pursue objectives in the same stages (listed above) as single agents.

There are several things to note here. First, the level of collaboration in the problem solving may vary greatly. In some cases, for example, the collaboration may be primarily in the planning phase, but one agent will actually execute the plan alone. In other cases, the collaboration may be active in all stages, including the planning and execution of a joint plan, where both agents execute actions in a coordinated fashion. Again, we want a model that will cover the range of possible levels of collaboration.

3 Problem-Solving Objects

The basic building blocks of our formal CPS model are problem-solving (PS) objects, which we represent as typed feature structures. PS object types form a single-inheritance hierarchy, where children inherit or specialize features from parents. Instances of these types are then used in problem solving.[1]

In our CPS model, we define types for the upper level of an ontology of PS objects, which we term *abstract PS objects*. These abstract PS objects are used to model problem-solving at a domain-independent level, and all operators (discussed below) operate on them. The model is then specialized to a domain by inheriting and instantiating domain-specific types and instances from the PS objects. The operators, however, do not change with domain, which allows reasoning to be done at a domain-independent level.

We first describe the abstract PS objects and then how they are specialized.

3.1 Abstract PS Objects

The following are the six abstract PS objects from which all other domain-specific PS objects inherit:

Objective A goal, subgoal or action. For example, in a rescue domain, objectives could include rescuing a person, evacuating a city, and so forth. We consider objectives to be actions rather than states, allowing us to unify the concepts of action and goal.

Recipe Beliefs of how to attain an objective. A recipe library can be expanded or modified through (collaborative or single-agent) planning.

Constraint A restriction on an object. Constraints are used to restrict possible solutions in the problem-solving process as well as possible referents in object identification.

Evaluation An assessment of an object's value within a certain problem-solving context. Agents will often evaluate several options before choosing one.

Situation The state of the world (or a possible world). In all but the simplest domains, agents may only have partial knowledge about a given situation.

Resource All other objects in the domain. These include include real-world objects (e.g., airplanes, ambulances) as well as concepts (e.g., song titles, artist names)

In order to ensure that all objects in our problem-solving model are labeled with a unique ID, we introduce a basic type *object* with a single attribute ID.[2] This is then used as the root of all objects in the hierarchy.

Each of the abstract PS objects share a set of common features. We put these common features in a new type, *ps-object*, which is the common parent of all of the abstract PS objects. We briefly describe its features here and then continue by giving the type declarations for each of the abstract PS objects in turn.

ps-object inherits from *object* and therefore contains an ID attribute (not shown — we will typically not list inherited features). It also has one additional attribute: CONSTRAINTS. This provides a way of describing the *ps-object* with a set of *constraints*.

It is important to note that the type of the CONSTRAINTS attribute is not simply a set of type *constraint*. Rather, it is one of a special class of middleman types we call *slots*. As these middleman types are a vital part of the CPS model, we take a brief aside here to discuss them before continuing with the abstract PS objects.

Slots and Fillers Collaborative problem solving can be seen as a decision-making process with respect to choosing and pursuing objectives. In modeling problem solving, we want to model more than just the decisions made; we want to model the decision-making *process* itself.

Within our model, decisions can be seen as the choosing of values (objects) or sets of values for certain roles. For example, agents decide on a set of objectives to pursue; for each objective they have, agents must decide on a (single) recipe to use in pursuing it; and so forth. A

[1] Due to space constraints, we omit here a discussion of the formal representation of objects. We refer readers to (Blaylock, 2005) for details.

[2] Formal definitions for all objects discussed here are found in Appendix A.

straightforward way of modeling these decisions would be to include, for example, a feature RECIPE in an objective which takes a recipe value, and represents the current recipe the agents are using to pursue this objective. Similarly, we could define a feature OBJECTIVES at the top level which would hold the set of objectives the agents are currently committed to.

Doing this, however, would only model the agents' *decision*, but not the *process* the agents followed/are following in making that decision. In deciding on a recipe to use for an objective, agents may identify several possible recipes as possibilities and evaluate each one. They may similarly narrow down the space of possible recipes by placing constraints on what they are willing to consider. These kinds of meta-decisions can constitute a large chunk of collaborative communication, but most models of dialogue do not represent them explicitly.

To be able to model these and other kinds of decisions-making processes, we add two levels of indirection at each decision point in the model. The first is what we call a *slot*, which contains information about the possible filler values (e.g., recipes) which have been/are under consideration in that context. A slot also contains information about possible constraints which have been put on what should be considered (e.g., not *all* valid recipes, but just those which take less than 30 minutes to execute). A slot also records which (if any) *filler* has been chosen by the agents.

A *filler* is the second layer of indirection. It is used to wrap an actual value with a set of evaluations the agents have made/might make about it. Note that this wrapping is necessary, as evaluations will always be context-dependent (i.e., dependent on the current slot) and cannot, therefore, be attached to the value itself.

Using these two levels of indirection gives us a rich model of not only the decisions (to be) made, but also the decision-making process itself., as we will show in the dialogue examples below.

As the parent of slots, we define an abstract *slot* type, which is the parent of *single-slot* and *multiple-slot*. These types differentiate decision points where just one filler is needed (e.g., a single recipe for an objective), or where a set of values can be chosen (e.g., objectives that the agents wish to pursue). We first discuss the single-value case, and then the multiple-value case.

Slots for Single Values The most typical case is where a single value can be used to fill a slot. *single-slot* is an abstract type for handling this.[3] It has three features (besides the ID inherited from *object*): IDENTIFIED is the set of all values (wrapped in *fillers*) that the agents have considered/are considering to fill this slot. ADOPTED records the single value which the agents have committed to for this slot. (This value may also be empty in the case that the agents have not yet made a decision, or have reversed a previous decision.) The CONSTRAINTS feature describes possible constraints the agents have put on possible slot fillers (such that the chosen recipe have 5 or fewer steps). Note that this is itself a type of slot, *constraints-slot*, which we will describe shortly.

[3]Note that this is actually a type schema, where σ can be instantiated with a given type. The same holds for the discussion of the *filler* type below.

Note also that the types of the IDENTIFIED and ADOPTED features contain a *filler* type. A *filler* contains both a VALUE which it wraps, as well an EVALUATIONS attribute, which represents any evaluations the agents may make/have made about the value in the local context. This is also a slot of type *evaluations-slot* which we will describe momentarily.

Slots for Sets of Values The type *multiple-slot* defines an abstract slot for decisions which allow more than one simultaneous value. In our model, we use three classes which inherit from *multiple-slot*: *constraints-slot*, *evaluations-slot*, and *objectives-slot*. We describe each in turn.

Previously-discussed types *ps-object* and *single-slot* have already introduced the *constraints-slot* type. As discussed above, this type allows a set of constraints to be identified and adopted in a context. Here the IDENTIFIED and ADOPTED features have a similar meaning to those in *single-slot*, with the only exception being that ADOPTED takes a set of *fillers*, instead of a single value.

The second slot type for multiple values is the *evaluations-slot*, which was used above in the definition of *filler*. An *evaluation-slot* provides a space for determining a set of *evaluations*. Its attributes are used in the same way to those of *single-slot* and *constraints-slot* and don't merit further comment here.

The final slot type for multiple values is the *objectives-slot*. It too has the features CONSTRAINTS, IDENTIFIED and ADOPTED which are used as they are in *evaluations-slot*. Objectives are not only committed to, but can also be executed. Objectives in the SELECTED set are those which the agents are currently executing (more details below), as opposed to just intending to execute. Finally, as discussed above, agents must monitor the situation in order to notice when an objective has been fulfilled (so that they stop pursuing it). Objectives which the agents believe have been fulfilled are put into the RELEASED set.

Objective Now that we have described the various slot and filler types which are used in the model, we are ready to get on with the definitions of the abstract PS object types. The type *objectives*, like all six abstract PS objects, inherits directly from *ps-object*. *objective* extends this by adding a RECIPE attribute which is of type *single-slot(recipe)*. This slot provides a place to track all problem-solving activity related to choosing a single *recipe* to use to pursue the *objective*, as discussed above.

Recipe Recipes are represented as a set of subobjectives (i.e., actions) and a set of constraints on those subobjectives. The ACTIONS attribute is an *objectives-slot* which allows a set of *objectives* associated with the *recipe*, as discussed above. The attribute ACTION-CONSTRAINTS contains the *constraints* placed on the *objectives*.

Constraint *Constraints* are represented as *boolean-expressions*. We do not define the form of these expressions here, but we envision a typical kind of expression involving boolean connectives (*and*, *or*, etc.) as well as (possibly domain-specific) predicates.

Resource *Resources* are used to represent what would typically be thought of as "objects" in a domain. These include real-world objects, but can also include any sort of object used in problem solving that does not fall into one of the other categories of abstract PS objects.

In addition to the attributes inherited from *ps-object*, *resources* contain the attribute ACTUAL-OBJECT, which holds a pointer to the "actual" object as represented in an agent's mental state.

Evaluation Before making decisions in problem solving, agents often evaluate each of the options that have been identified. An *evaluation* represents the agents' assessment of a particular PS object within a particular context. The *evaluation* is therefore always associated with a PS object and a context (e.g., which PS object to choose to fill a slot). As we are not yet sure how best to represent the evaluations themselves, we leave the type of the ASSESSMENT attribute unstructured.

Situation A *c-situation*,[4] describes the state of a possible world, or more precisely, the agents' beliefs about that possible world. Rather than just packing all state information into a general world-state attribute, we separate out information about problem-solving in the situation, and then have a separate place to store other world beliefs.

The PS-OBJECTS attribute holds a set of all PS objects known to the agents in the situation, whereas PENDING-PS-OBJECTS is a temporary holder for objects which have not yet been successfully identified (but have been mentioned in the conversation). These sets include domain-specific PS objects (such as objectives, recipes and resources) which the agents can use in problem solving.

The PS-HISTORY attribute records the history of the agents' problem solving. This is a list of interaction acts the agents have performed (see below).

The OBJECTIVES attribute holds the *objectives* they agents are considering/planning/executing in the situation.

The FOCUS attribute tracks the agents' problem-solving focus as a stack, which is similar to linguistic focus in (Grosz and Sidner, 1986).

All other beliefs about the world are stored in the CONSTRAINTS attribute (inherited from *ps-object*). This models the fact that the agents do not have perfect knowledge of the world state, but rather only bits of knowledge which constrain which state it is actually in.

[4]This object as well as the CPS acts below are prefaced with *c-* to differentiate them with types in our model of single-agent problem solving, which is not discussed here.

3.2 Domain Specialization

The CPS model can be specialized to a domain by creating new types that inherit from the abstract PS objects and/or creating instantiations of them. We describe each of these cases separately.

Specialization through Inheritance As described above, inheritance is basically the process of adding new attributes to a previously existing type, and/or specializing the types of preexisting attributes. In our CPS model, inheritance is only used for *objectives*, and *resources*. The other abstract PS objects are specialized through instantiation.

Inheriting from *resource* is done to specify domain-specific resource type. As an example, we define a *ship-by-train* objective for a logistics domain (used in an example below), which is also shown in Appendix A. This inherits all attributes from *objective* and adds two more: the item to be shipped and a destination which are *single-slots* of type *movable* and *city* respectively (definitions not shown).

Specialization through Instantiation All PS object types (including new types created by inheritance) can be further specialized by instantiation, i.e., by assigning values to some set of their attributes. This can be done both at design time (by the domain modeler) and (as we discuss below) it happens at runtime as part of the problem-solving process itself.

4 The Collaborative Problem Solving State

The CPS state is part of the agents' common ground (Clark, 1996), and models the agents' current problem-solving context. It is represented as an instance of type *c-situation* called the *actual-situation*. As the name implies, the *actual-situation* is a model of the agents' beliefs about the current situation and the actual problem-solving context.

The OBJECTIVES attribute contains all of the top-level *objectives* associated with the agents' problem solving process. These *objectives* form the roots of individual problem-solving contexts associated with reasoning with, and/or trying to accomplish those *objectives*, and can include all types of other PS objects.

5 Collaborative Problem Solving Acts

Agents change their CPS state through the execution of CPS acts. There are two broad categories of CPS acts: those used in reasoning and those used for commitment. We describe several *families* of CPS act types within those categories:

Reasoning Act Families

- *c-focus*: Used to focus problem solving on a particular *object*.
- *c-defocus*: Removes the focus on a particular *object*.
- *c-identify*: Used to identify a *ps-object* as a possible option in a certain context.

Commitment Act Families

- *c-adopt*: Commits the agents to an *object* in a certain context.
- *c-abandon*: Removes an existing commitment to an *object*.
- *c-select*: Moves an *objective* into active execution.
- *c-defer*: Removes an *objective* from active execution (but does not remove a commitment to it).
- *c-release*: Removes the agents' commitment to an *objective* which they believe has been fulfilled.

Each of these families encompasses a set of CPS acts. For the remainder of this section, we discuss each of the CPS act families and their corresponding acts as well as their effects on the CPS state.

5.1 c-focus/c-defocus

Agents need to coordinate their problem-solving focus. They do this through the execution of the following CPS acts:

c-focus(situation-id,object-id)
c-defocus(situation-id,object-id)

The semantics of these are simple. *c-focus* pushes the given *object-id* onto the focus stack in the *c-situation* represented by *situation-id*. *c-defocus* pops the *object-id* off the stack as well as any *object-ids* above it.

5.2 c-identify

CPS acts in the *c-identify* family are used to introduce CPS objects into the realm of a problem-solving context. This could either be in identifying previously unknown objects (i.e., objects not listed in PS-OBJECTS within the *c-situation*), or it could be in identifying a known object as a possible option for filling a certain slot.

All objects must be identified before they can be used further in the CPS process. For this reason, CPS acts in the *c-identify* family exist for all PS objects.[5] The are as follows: *c-identify-objective*, *c-identify-recipe*, *c-identify-constraint*, *c-identify-resource*, and *c-identify-evaluation*.

The basic syntax of identify acts is

c-identify-{type}(slot-id,ps-object)

where *type* refers to any of the PS objects in the acts listed above. The *ps-object* parameter is the PS object instance which is being introduced and the *slot-id* parameter gives the *id* of the problem-solving context for which it is being identified. Note that *c-identify* acts take an actual *ps-object* as an argument, whereas the remaining CPS acts take only an *object-id* (i.e., pointer to an object).

The effect of a *c-identify* is that the *ps-object* is inserted into the PS-OBJECTS set in the *actual-situation* (if not already there). It is also wrapped in an appropriate *filler* type and inserted into the IDENTIFIED set of the *slot* identified by *slot-id*.

5.3 c-adopt/c-abandon

We treat the CPS act families *c-adopt* and *c-abandon* together here, as one essentially undoes the other. The syntax of the two is as follows:

c-adopt-{type}(slot-id,filler-id)
c-abandon-{type}(slot-id,filler-id)

As with *c-identify*, these two families have types corresponding to most abstract PS objects: *c-adopt-objective*, *c-adopt-recipe*, *c-adopt-resource*, *c-adopt-constraint*, *c-adopt-evaluation*, *c-abandon-objective*, *c-abandon-recipe*, *c-abandon-resource*, *c-abandon-constraint*, and *c-abandon-evaluation*.

A *c-adopt* has the effect of adding the *filler* referred to by *filler-id* to the ADOPTED attribute of the *slot* referred to by *slot-id* (either by assigning the value, in the case of a *single-slot* or adding the value to the set for a *multiple-slot*. Note that this requires that the PS object referred to by *filler-id* be in the IDENTIFIED set in that context.

A *c-abandon* basically has the opposite effect. It removes the object from the ADOPTED attribute. Thus *c-abandon* requires that the object actually be adopted when the act is executed.

When a PS object is adopted with respect to a slot, it means the agents are committed to that object in that context. For example, for a *recipe*, this means the agents are committed to using that *recipe* for the associated *objective*. The other PS objects are similarly treated.

[5]Except *c-situation* for reasons described above.

5.4 c-select/c-defer

The act families *c-select* and *c-defer* are only used for *objectives*. Their syntax is as follows:

c-select-objective(slot-id,filler-id)
c-defer-objective(slot-id,filler-id)

Executing a *c-select-objective* adds an *objective* to the SELECTED set in the given *objectives-slot*. *c-defer-objective* can then be used to delete an object from the SELECTED set.

Although agents may have any number of adopted *objectives*, there is only a small subset that is actually being executed at any given point. These are the *objectives* in the SELECTED set. An *objective* does not need to be an atomic action to be selected. Higher-level *objectives* can be marked as selected if the agents believe that they are currently executing some action as part of executing the higher-level *objective*.

5.5 c-release

The final CPS act we discuss here is *c-release*. As with *c-select* and *c-defer*, this is only applicable to *objectives*. The syntax is as follows:

c-release-objective(slot-id,filler-id)

This act has the effect of moving an *objective* filler from the ADOPTED set to the RELEASED set. Note that the *objective* must first be in the ADOPTED set for this act to be executed.

Rational agents should notice when an *objective* has been successfully achieved and then stop intending to achieve it (cf. (Cohen and Levesque, 1990)). The RELEASED set contains those *objectives* which the agents believe have successfully been achieved.

6 Interaction Acts

An agent cannot single-handedly execute CPS acts to make changes to the CPS state. Doing so requires the cooperation and coordination of both agents. In the model, CPS acts are generated by sets of *interaction acts* (IntActs) — actions that single agents execute in order to negotiate and coordinate changes to the CPS state. An IntAct is a single-agent action which takes a CPS act as an argument.

The IntActs are *begin*, *continue*, *complete* and *reject*. An agent beginning a new CPS act proposal performs a *begin*. For successful generation of the CPS act, the proposal is possibly passed back and forth between the agents, being revised with *continues*, until both agents finally agree on it, which is signified by an agent *not* adding any new information to the proposal but simply accepting it with a *complete*. This generates the proposed CPS act resulting in a change to the CPS state. At any point in this exchange, either agent can perform a *reject*, which causes the proposed CPS act — and thus the proposed change to the CPS state — to fail.

7 Grounding Acts

The model as it stands thus far makes the simplifying assumption that utterances are always correctly heard by the hearer and that he also correctly interprets them (i.e., properly recovers the intended (instantiated) interaction acts). In human communication, mishearing and misunderstanding can be the rule, rather than the exception. Because of this, both speaker and hearer need to *collaboratively* determine the meaning of an utterance through a process termed *grounding* (Clark, 1996).

We add grounding to our model by utilizing the *Grounding Acts* (GAs) proposed as part of Conversation Acts theory (Traum and Hinkelman, 1992)[6] and used in Traum's computational model of grounding (Traum, 1994). In our model, we expand the definition of GAs to allow them to take individual IntActs as arguments. As the grounding acts themselves are not our focus here, we discuss them only briefly. The Grounding Acts are as follows:

Initiate The initial contribution of an IntAct.

Continue Used when the initiating agent has a turn of several utterances. An utterance which further expands the meaning of the IntAct.

Acknowledge Signals understanding of the IntAct (although not necessarily *agreement*, as this is modeled at the IntAct level).

Repair Changes some part of the IntAct.

ReqRepair A request that the other agent repair the IntAct.

ReqAck An explicit request for an acknowledgment by the other agent.

Cancel Declares the attempted IntAct as 'dead' and ungrounded.

In the model, an IntAct is not successfully executed until it has been successfully grounded. This is typically after an *acknowledge*, although see (Traum, 1994) for details.

8 Example

To show concretely how the model is used, we analyze and discuss here a human-human dialogue from

[6]Interestingly enough, our interaction acts and CPS acts could be seen as roughly corresponding to the Core Speech Acts and Argumentation Acts levels in Conversation Acts theory.

(Traum and Hinkelman, 1992). As a further example, Appendix C shows the analysis for a human-human dialogue from (Grosz and Sidner, 1986), although space precludes a detailed discussion of the analysis.[7]

Figure 1 shows the dialogue from (Traum and Hinkelman, 1992) marked up with instantiated grounding acts.[8] We first discuss the dialogue at the grounding level, and then at the problem-solving level.

Grounding Our analysis at the grounding level is basically unchanged from that of (Traum and Hinkelman, 1992). We therefore describe it only briefly. The main difference between the two accounts is that we associate GAs with individual IntActs, whereas they associate them with utterances.

In Utterance Unit (UU) 1.1, the user initiates three IntActs, which the system acknowledges in UU 2.1. (Note that, for compactness, we use subscripted ranges to refer to series of GAs and IntActs. ack_{1-2} expands to ack_1 and ack_2, and $\mathit{init}_{3-4}(\mathit{complete}_{1-2})$ expands to $\mathit{init}_3(\mathit{complete}_1)$ and $\mathit{init}_4(\mathit{complete}_2)$.) Note that, only after UU 2.1 are the are the effects of the IntActs 1–3 valid and cause a change to the CPS state.

UU 2.1 also initiates the corresponding *complete* IntActs to those initiated in UU 1.1; these are acknowledged in UU 3.1.

UU 3.1 also inits three IntActs (7, 8, and 9), which are never grounded.[9] These, therefore don't result in any successfully executed IntActs, and thus no changes to the CPS state.

In UU 3.2, the user inits new IntActs for which he explicitly requests an acknowledgment in UU 3.3. UU 4.1 inits two completes, which are acknowledged by the subsequent utterance by the user (not shown).

The remainder of the dialogue consists solely of paired inits and acks, which are handled similarly to the first two utterances.

Problem Solving At the problem-solving level, in UU 1.1., the user proposes the adoption (and identification) of an objective of shipping oranges. (Again, these do not become valid until grounded in UU 2.1). He also proposes that problem-solving focus be placed on that objective (i.e., in order to work on accomplishing it). The proposed objective is shown in Figure 2, and deserves some explanation.

The type of the objective is *ship-by-train*, which we have invented for this example.[10] It introduces two new attributes to the *objective* class: an item to be shipped and a destination. As the abbreviated form of the objective shows,[11] there are three main components to the objective as it has been introduced by the user. First of all, it has a pre-adopted destination — Bath (modeled as a location with a NAME). Second, the item to be shipped has not yet been determined, but a constraint has been put on possible values for that slot — they must be of type *oranges*.

Recall that this was one of the motivations for introducing slots in the model. Notice here that the constraint is not put on a particular instance of oranges, rather it is put on the *single-slot* itself. Constraints on a slot are adopted to restrict the values considered (e.g., identified) as possible fillers.

Finally, a constraint has also been placed on the objective itself — that it be completed by 8am. Note the difference here between placing a constraint on a *ps-object* versus placing it on a *slot*, as just discussed.

UU 2.1 completes the CPS acts, and after 3.1, when the completes are grounded, the CPS acts are generated, resulting in the corresponding changes to the CPS state.

In UU 3.1, the user proposes the adoption of a (partial) recipe for shipping the oranges, as well as that focus be placed on it. These IntActs are never grounded, and thus never result in a change in the CPS state. However, as this utterance gives a good example of the introduction of a recipe, we still discuss it.

The recipe that the user attempts to introduce is shown in Figure 3, and consists of a single adopted objective — that of moving a boxcar to Corning. Similar to the *ship-by-train* objective above, this also has an adopted value (the destination is Corning), and a constraint on the slot of the other (that the only resources to be considered to be moved should be of type *boxcar*). At this point, the recipe has no ACTION-CONSTRAINTS.

As mentioned, however, this recipe never makes it into the CPS state. Instead, in UU 3.2, the user decides he wants to adopt (confirm) the joint belief that there are oranges at Corning. Note that no focus change is proposed by UU 3.2. We model this in this way, as it appears that the user did not intend for further work (beyond adoption) to be done on this constraint, or on finding out the state of the world. Instead, it was intended as a quick check, but focus was intended to remain on the *ship-by-train* objective.

As discussed above, we model beliefs about the world

[7]The description of this analysis as well as other examples can be found in (Blaylock, 2005).

[8]We have at times combined multiple utterance units into a single utterance unit in the case that the units were tagged as *conts* at the grounding level. Although modeling this is important at the grounding level, an *init* followed by *conts* are just gathered up into a single (group of) IntActs that are proposed, so at the problem-solving level, this is not an important difference.

[9]Although see discussion in (Traum and Hinkelman, 1992).

[10]In all examples, we invent simple-minded domain-specific object types as we need them.

[11]Because of space constraints, we only list the most salient features of the objective, given with their feature paths. We also abbreviate feature names as detailed in Appendix B.

1.1 U: okay, the problem is we better ship a boxcar of oranges to Bath by 8 AM.
init$_1$(*begin*$_1$(*c-identify-objective*(STATE | OBJVS | ID, $\boxed{1}$)))
init$_2$(*begin*$_2$(*c-adopt-objective*(STATE | OBJVS | ID, $\boxed{1}$ | ID)))
init$_3$(*begin*$_3$(*c-focus*(STATE | ID, $\boxed{1}$ | ID)))
2.1 S: okay.
ack$_{1-3}$
init$_{4-6}$(*complete*$_{1-3}$)
3.1 U: now ... umm ... so we need to get a boxcar to Corning, where there are oranges.
ack$_{4-6}$
init$_7$(*begin*$_4$(*c-identify-recipe*($\boxed{1}$ | REC | ID, $\boxed{2}$)))
init$_8$(*begin*$_5$(*c-adopt-recipe*($\boxed{1}$ | REC | ID, $\boxed{2}$ | ID)))
init$_9$(*begin*$_6$(*c-focus*(STATE | ID, $\boxed{2}$ | ID)))
3.2 U: there are oranges at Corning
init$_{10}$(*begin*$_7$(*c-identify-constraint*(STATE | CONS | ID, $\boxed{3}$)))
init$_{11}$(*begin*$_8$(*c-adopt-constraint*(STATE | CONS | ID, $\boxed{3}$ | ID)))
3.3 U: right?
reqack$_{10-11}$
4.1 S: right.
ack$_{10-11}$
init$_{12-13}$(*complete*$_{7-8}$)
⋮ (Utterances 5.1–13.1 removed due to space constraints)
13.2 U: or, we could actually move it [Engine E1] to Dansville, to pick up the boxcar there
init$_1$(*begin*$_1$(*c-identify-recipe*($\boxed{1}$ | REC | ID, $\boxed{4}$ *[move(E1,Dansville)]*)))
init$_2$(*begin*$_2$(*c-focus*(STATE | ID, $\boxed{4}$ | ID)))
14.1 S: okay.
ack$_{1-2}$
init$_{3-4}$(*complete*$_{1-2}$)
15.1 U: um and hook up the boxcar to the engine, move it from Dansville to Corning, load up some oranges into the boxcar, and then move it on to Bath.
ack$_{3-4}$
init$_5$(*begin*$_3$(*c-identify-objective*($\boxed{4}$ | ACTS | ID, $\boxed{5}$ *[hook(boxcar1,engine1)]*))
init$_6$(*begin*$_4$(*c-adopt-objective*($\boxed{4}$ | ACTS | ID, $\boxed{5}$ | ID)))
init$_7$(*begin*$_5$(*c-identify-constraint*($\boxed{4}$ | ACONS | ID, $\boxed{6}$ *[before(*$\boxed{1}$, $\boxed{2}$*)]*)))
init$_8$(*begin*$_6$(*c-adopt-constraint*($\boxed{4}$ | ACONS | ID, $\boxed{6}$ | ID))
init$_{9-16}$(*begin*$_{7-14}$) *[2 other actions and 2 other ordering constraints]*
16.1 S: okay.
ack$_{5-16}$
init$_{17-28}$(*complete*$_{3-14}$)
17.1 U: how does THAT sound?
ack$_{17-28}$
init$_{29}$(*begin*$_{15}$(*c-identify-evaluation*(FILLER($\boxed{4}$) | EVALS | ID, $\boxed{7}$ *[blank evaluation]*)))
18.1 S: that gets us to Bath at 7 AM, and (inc) so that's no problem.
ack$_{29}$
init$_{30}$(*continue*$_{15}$(*c-identify-evaluation*(FILLER($\boxed{4}$) | EVALS | ID, $\boxed{7}$ *[sufficient]*)))
init$_{31}$(*begin*$_{16}$(*c-adopt-evaluation*(FILLER($\boxed{4}$) | EVALS | ID, $\boxed{7}$ | ID)))
19.1 U: good.
ack$_{30-31}$
init$_{32-33}$(*complete*$_{15-16}$)
init$_{34}$(*begin*$_{17}$(*c-adopt-recipe*($\boxed{1}$ | REC | ID, $\boxed{4}$ | ID))
init$_{35}$(*begin*$_{18}$(*c-defocus*(STATE | ID, $\boxed{4}$ | ID)))
20.1 S: okay.
ack$_{32-35}$
init$_{36-37}$(*complete*$_{17-18}$)

Figure 1: Analysis of a Planning Dialogue from (Traum and Hinkelman, 1992)

$$\boxed{1}\ {}_{\textit{ship-by-train}}\left[\begin{array}{ll} \text{CONS}\,|\,\text{APTD} & \left\{ {}_{\textit{fill}}\left[\text{VAL}\,|\,\text{EXPR} \quad \textit{'completion before 8am'}\right]\right\} \\ \text{ITEM}\,|\,\text{CONS}\,|\,\text{APTD} & \left\{ {}_{\textit{fill}}\left[\text{VAL}\,|\,\text{EXPR} \quad \textit{'type = oranges'}\right]\right\} \\ \text{DEST}\,|\,\text{APTD} & {}_{\textit{fill}}\left[\text{VAL}\,|\,\text{NAME} \quad \textit{'Bath'}\right] \end{array}\right]$$

Figure 2: Contents of *objective* $\boxed{1}$

$$\boxed{2}\ {}_{\textit{rec}}\left[\text{ACTS}\,|\,\text{APTD} \quad \left\{ {}_{\textit{fill}}\left[\text{VAL} \quad {}_{\textit{move}}\left[\begin{array}{ll} \text{ITEM}\,|\,\text{CONS}\,|\,\text{APTD} & \left\{ {}_{\textit{fill}}\left[\text{VAL}\,|\,\text{EXPR} \quad \textit{'type = boxcar'}\right]\right\} \\ \text{DEST}\,|\,\text{APTD}\,|\,\text{VAL}\,|\,\text{NAME} & \textit{'Corning'} \end{array}\right]\right]\right\}\right]$$

Figure 3: Contents of *recipe* $\boxed{2}$

state as constraints on the situation. The proposed constraint is shown in Figure 4. In this paper, we do not present a theory of constraint representation, thus we have been glossing constraints until now. The only specification we have made is that the EXPRESSION be of type *boolean*. In the case of this constraint, however, a simple gloss is not enough, as this constraint actually introduces a new embedded *resource* — the instance of the oranges that are at Corning. For this reason, we show this constraint as a domain-specific predicate (*at*) that takes a location and an item. More work obviously needs to be done on the general specification of constraints, but the representation here is sufficient for our purposes.

It is important to point out that when the *beginc-identify-constraint* is grounded in UU 4.1, the oranges instance from the constraint is also placed in the PS-OBJECTS set within the CPS state, making it available for use in further problem solving.

In the interest of space and clarity, we have skipped part of the dialogue (Utterance Units 5.1–13.1) which included mostly grounding interaction which is adequately described in (Traum and Hinkelman, 1992). In UU 13.2, the user identifies a possible (partial) recipe for the ship-by-train objective. The recipe includes a single objective (action) of moving engine E1 to Dansville.[12] Note that the user does not propose that this recipe be adopted for the objective, yet; he only proposes that it be considered as a candidate. He also proposes moving problem-solving focus to the recipe (in order to work on expanding it).

In 14.1, the system acknowledges these grounding acts and also inits IntActs to complete them. In 15.1, the user proposes several new actions to add to the recipe as well as ordering constraints among them. In UU 17.1, the user

[12]From now on, we will gloss PS objects with a boxed number, $\boxed{4}$ in this case, and some description of their contents.

$$\boxed{3}\ {}_{\textit{con}}\left[\text{EXPR} \quad \textit{at}\left({}_{\textit{loc}}\left[\text{NAME} \quad \textit{'Corning'}\right], {}_{\textit{oranges}}\left[\text{AOBJ} \quad \textit{oranges37}\right]\right)\right]$$

Figure 4: Contents of *constraint* $\boxed{3}$

then asks for the system's evaluation of the recipe, which is provided in UU 18.1. The surface form of 19.1 is a bit misleading. With this "good", the user is acking the system's last utterance and accepting the given evaluation. He is also, based on this evaluation, proposing that the recipe be adopted for the objective and proposing that the focus be taken off the recipe. The system accepts these proposals in the final utterance.[13]

9 Related Work

The work in (Cohen et al., 1991) motivates dialogue as the result of the intentions of rational agents executing joint plans. Whereas their focus was the formal representation of single and joint intentions, we focus on describing and formalizing the interaction itself. We also extend coverage to the entire problem-solving process, including goal selection, planning, and so forth.

Our work is also similar in spirit to work on SharedPlans (Grosz and Sidner, 1990; Grosz and Kraus, 1996), which describes the necessary intentions for agents to build and hold a joint plan, as well as a high-level sketch of how such joint planning occurs. It defines four operators which describe the planning process: *Select_Rec*, *Elaborate_Individual*, *Select_Rec_GR*, and *Elaborate_Group*. Our CPS acts describe the joint

[13]As with any grounding model, we have the problem that final utterances will not be verbally acked. We assume within the model that, if after a small pause, if there is no evidence of the user *not* having heard or understood, inits of completes are considered acknowledged.

planning process at a more fine-grained level in order to be able to describe contributions of individual utterances. The CPS acts could possibly be seen as a further refinement of the SharedPlans operators. Our model also describes other problem-solving stages, such as joint execution and monitoring.

Collagen (Rich et al., 2001) is a framework for building intelligent interactive systems based on Grosz and Sidner's tripartite model of discourse (Grosz and Sidner, 1986). It provides middleware for creating agents which act as collaborative partners in executing plans using a shared artifact (e.g., a software application). In this sense, it is similar to the work of Cohen and Levesque described above.

Collagen uses a subset of Sidner's artificial negotiation language (Sidner, 1994) to model individual contributions of utterances to the discourse state. The language defines operators with an outer layer of negotiation (e.g., *ProposeForAccept* (*PFA*) and *AcceptProposal* (*AP*)) which take arguments such as *SHOULD(action)* and *RECIPE*. Our interaction and collaborative problem-solving acts are similar in spirit to Sidner's negotiation language, covering a wider range of phenomena in more detail (including evaluations of goals and recipes, solution constraining, and a layer of grounding).

10 Conclusion and Future Work

In this paper, we have presented a formal model of agent collaborative problem solving. We also introduced a model of dialogue created by combining the CPS model with a known model of grounding. The dialogue model is able to support a wide range of dialogue phenomena, including dialogues supporting interleaved planning and execution, mixed-initiative dialogues, and grounding.

We are currently developing an autonomous agent dialogue manager that can participate in collaborative problem solving, which serves as the back-end of a dialogue system. In addition, we are developing a generation component that can generate multimodal output from these instantiated grounding acts. Finally, we plan to develop an intention recognizer that is capable of recognizing these instantiated grounding acts.

Acknowledgments

We would like to thank George Ferguson for early collaboration on the CPS model and the members of the Saarbrücken TALK group (especially Ivana Kruijff-Korbayová and Tilman Becker) for input on the model.

This material is based upon work supported by a grant from DARPA under grant number F30602-98-2-0133; two grants from the National Science Foundation under grant number IIS-0328811 and grant number E1A-0080124; and the EU-funded TALK project (No. IST-507802). 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 above-mentioned organizations.

References

James F. Allen, Donna K. Byron, Myroslava Dzikovska, George Ferguson, Lucian Galescu, and Amanda Stent. 2001. Towards conversational human-computer interaction. *AI Magazine*, 22(4):27–37.

James Allen, Nate Blaylock, and George Ferguson. 2002. A problem-solving model for collaborative agents. In *AAMAS'02*, Bologna, Italy, July 15-19.

Nate Blaylock, James Allen, and George Ferguson. 2003. Managing communicative intentions with collaborative problem solving. In *Current and New Directions in Discourse and Dialogue*. Kluwer, Dordrecht.

Nathan J. Blaylock. 2005. *Towards Tractable Agent-based Dialogue*. Ph.D. thesis, University of Rochester, Dept. of Computer Science.

Herbert H. Clark. 1996. *Using Language*. Cambridge University Press.

Philip R. Cohen and Hector J. Levesque. 1990. Intention is choice with commitment. *Artificial Intelligence*, 42:213–261.

Philip R. Cohen, Hector J. Levesque, José H. T. Nunes, and Sharon L. Oviatt. 1991. Task-oriented dialogue as a consequence of joint activity. In *Artificial Intelligence in the Pacific Rim*. IOS Press, Amsterdam.

Barbara J. Grosz and Sarit Kraus. 1996. Collaborative plans for complex group action. *Artificial Intelligence*, 86(2):269–357.

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

Barbara J. Grosz and Candace L. Sidner. 1990. Plans for discourse. In *Intentions in Communication*. MIT Press, Cambridge.

Charles Rich, Candace L. Sidner, and Neal Lesh. 2001. COLLAGEN: Applying collaborative discourse theory to human-computer interaction. *AI Magazine*, 22(4):15–25.

Candace L. Sidner. 1994. An artificial discourse language for collaborative negotiation. In *AAAI*, pages 814–819, Seattle, WA.

David R. Traum and Elizabeth A. Hinkelman. 1992. Conversation acts in task-oriented spoken dialogue. *Computational Intelligence*, 8(3):575–599.

David R. Traum. 1994. *A Computational Theory of Grounding in Natural Language Conversation*. Ph.D. thesis, University of Rochester, Department of Computer Science.

A Definitions of PS Objects

The following are the feature structure type definitions for the objects described in this paper (including the domain-specific types used in the example). Type name and parent class are shown above the feature structure in the form **type** ← **parent**.

object ← ϵ

$$\begin{bmatrix}\textsc{id} & \textit{id}\end{bmatrix}$$

slot ← **object**

$$\begin{bmatrix}\textsc{identified} & \textit{set(filler(ps-object))}\end{bmatrix}$$

single-slot(σ) ← **slot**

$$\begin{bmatrix}\textsc{constraints} & \textit{constraints-slot}\\ \textsc{identified} & \textit{set(filler}(\sigma)\textit{))}\\ \textsc{adopted} & \textit{filler}(\sigma)\end{bmatrix}$$

filler(σ) ← **object**

$$\begin{bmatrix}\textsc{evaluations} & \textit{evaluations-slot}\\ \textsc{value} & \sigma\end{bmatrix}$$

multiple-slot ← **slot**

$$\begin{bmatrix}\textsc{identified} & \textit{set(filler(ps-object))}\\ \textsc{adopted} & \textit{set(filler(ps-object))}\end{bmatrix}$$

constraints-slot ← **multiple-slot**

$$\begin{bmatrix}\textsc{identified} & \textit{set(filler(constraint))}\\ \textsc{adopted} & \textit{set(filler(constraint))}\end{bmatrix}$$

evaluations-slot ← **multiple-slot**

$$\begin{bmatrix}\textsc{constraints} & \textit{constraints-slot}\\ \textsc{identified} & \textit{set(filler(evaluation))}\\ \textsc{adopted} & \textit{set(filler(evaluation))}\end{bmatrix}$$

objectives-slot ← **multiple-slot**

$$\begin{bmatrix}\textsc{constraints} & \textit{constraints-slot}\\ \textsc{identified} & \textit{set(filler(objective))}\\ \textsc{adopted} & \textit{set(filler(objective))}\\ \textsc{selected} & \textit{set(filler(objective))}\\ \textsc{released} & \textit{set(filler(objective))}\end{bmatrix}$$

ps-object ← **object**

$$\begin{bmatrix}\textsc{constraints} & \textit{constraints-slot}\end{bmatrix}$$

objective ← **ps-object**

$$\begin{bmatrix}\textsc{recipe} & \textit{single-slot(recipe)}\end{bmatrix}$$

recipe ← **ps-object**

$$\begin{bmatrix}\textsc{actions} & \textit{objectives-slot}\\ \textsc{action-constraints} & \textit{constraints-slot}\end{bmatrix}$$

resource ← **ps-object**

$$\begin{bmatrix}\textsc{actual-object} & \textit{id}\end{bmatrix}$$

constraint ← **ps-object**

$$\begin{bmatrix}\textsc{expression} & \textit{boolean-expression}\end{bmatrix}$$

evaluation ← **ps-object**

$$\begin{bmatrix}\textsc{assessment} & \textit{unstructured}\end{bmatrix}$$

c-situation ← **ps-object**

$$\begin{bmatrix}\textsc{pending-ps-objects} & \textit{set(ps-object)}\\ \textsc{ps-objects} & \textit{set(ps-object)}\\ \textsc{ps-history} & \textit{list(interaction-act)}\\ \textsc{focus} & \textit{stack(object)}\\ \textsc{objectives} & \textit{objectives-slot}\end{bmatrix}$$

ship-by-train ← **objective**

$$\begin{bmatrix}\textsc{item} & \textit{single-slot(moveable)}\\ \textsc{dest} & \textit{single-slot(city)}\end{bmatrix}$$

B Abbreviations Used

The following table shows the various abbreviations of type and feature names used within the examples.

types		**features**	
full	**abbr**	**full**	**abbr**
object	obj	constraints	cons
ps-object	psobj	identified	ided
single-slot	ss	adopted	aptd
filler	fill	evaluations	evals
constraints-slot	cslot	value	val
evaluations-slot	eslot	selected	seld
objectives-slot	oslot	released	reld
objective	objv	recipe	rec
recipe	rec	actions	acts
constraint	con	action-constraints	acons
resource	res	expression	exp
evaluation	eval	actual-object	aobj
situation	sit	ps-objects	psobjs
c-situation	csit	ps-history	pshist
		objectives	objvs
		pending-ps-objects	pend

C Analysis of an Expert-Apprentice Dialogue from (Grosz and Sidner, 1986)

The context at the start of this segment is that the expert is specifying a recipe $\boxed{rec}$ to the apprentice for removing a pump. A description of the analysis (as well as other examples) can be found in (Blaylock, 2005)

1.1 E: First you have to remove the flywheel
init$_1$*(begin*$_1$*(c-identify-objective(* $\boxed{rec}$ | ACTS | ID, $\boxed{1}$ *[remove(flywheel)]))*
init$_2$*(begin*$_2$*(c-adopt-objective(* $\boxed{rec}$ | ACTS | ID, $\boxed{1}$ | ID*)))*
init$_3$*(begin*$_3$*(c-select-objective(* $\boxed{rec}$ | ACTS | ID, $\boxed{1}$ | ID*)))*

2.1 A: How do I remove the flywheel?
ack$_{1-3}$ *init*$_{4-6}$*(complete*$_{1-3}$*)*
init$_7$*(begin*$_4$*(c-focus(*STATE | ID, $\boxed{1}$ | REC | ID*)))*
init$_8$*(begin*$_5$*(c-identify-recipe(* $\boxed{1}$ | REC | ID, $\boxed{2}$ *[blank recipe])))*
init$_9$*(begin*$_6$*(c-adopt-recipe(* $\boxed{1}$ | REC | ID, $\boxed{2}$ | ID*)))*

3.1 E: First, loosen the two allen head setscrews holding it to the shaft, then pull it off
ack$_{4-9}$ *init*$_{10}$*(complete*$_4$*)*
init$_{11}$*(continue*$_5$*(c-identify-recipe(* $\boxed{1}$ | REC | ID, $\boxed{2}$ *[loosen(screwsets),pull-off(wheel)])))*
init$_{12}$*(continue*$_6$*(c-adopt-recipe(* $\boxed{1}$ | REC | ID, $\boxed{2}$ | ID*)))*

4.1 A: OK.
ack$_{10-11}$ *init*$_{12-13}$*(complete*$_{5-6}$*)*

⋮ (Utterances 5.1–17.1 removed due to space constraints)

18.1 A: The two screws are loose,
init$_{14}$*(begin*$_7$*(c-release-objective(* $\boxed{rec}$ | ACTS | ID, $\boxed{loosen}$ | ID*))*

18.2 A: but I'm having trouble getting the wheel off
init$_{15}$*(begin*$_8$*(c-select-objective(* $\boxed{rec}$ | ACTS | ID, $\boxed{pull-off}$ | ID*)))*
init$_{16}$*(begin*$_9$*(c-focus(*STATE | ID, $\boxed{pull-off}$ | REC | ID*)))*
init$_{17}$*(begin*$_{10}$*(c-identify-recipe(* $\boxed{pull-off}$ | REC | ID, $\boxed{3}$ *[blank recipe])))*
init$_{18}$*(begin*$_{11}$*(c-adopt-recipe(* $\boxed{pull-off}$ | REC | ID, $\boxed{3}$ | ID*)))*

19.1 E: Use the wheelpuller.
ack$_{14-18}$ *init*$_{19-21}$*(complete*$_{7-9}$*)*
init$_{22}$*(continue*$_{10}$*(c-identify-recipe(* $\boxed{pull-off}$ | REC | ID, $\boxed{3}$ *[recipe using wheelpuller])))*
init$_{23}$*(continue*$_{11}$*(c-adopt-recipe(* $\boxed{pull-off}$ | REC | ID, $\boxed{3}$ | ID*)))*

19.2 E: Do you know how to use it?
reqack$_{22-23}$

20.1 A: No.
ack$_{19-21}$ *reqrepair*$_{22-23}$

21.1 E: Do you know what it looks like?
cancel$_{22-23}$
init$_{24}$*(begin*$_{12}$*(c-identify-resource(*STATE | ID, $\boxed{4}$ *[wheelpuller])))*

22.1 A: Yes.
ack$_{24}$ *init*$_{25}$*(continue*$_{12}$*)*

23.1 E: Show it to me please.
ack$_{25}$ *init*$_{26}$*(continue*$_{12}$*)*
init$_{27}$*(begin*$_{13}$*(c-identify-objective(*STATE | OBJVS | ID, $\boxed{5}$ *[show(A,E,* $\boxed{4}$ *)])))*
init$_{28}$*(begin*$_{14}$*(c-adopt-objective(*STATE | OBJVS | ID, $\boxed{5}$ | ID*)))*
init$_{29}$*(begin*$_{15}$*(c-select-objective(*STATE | OBJVS | ID, $\boxed{5}$ | ID*)))*

24.1 A: OK.
ack$_{26-29}$ *init*$_{30}$*(continue*$_{12}$*)* *init*$_{31-33}$*(complete*$_{13-15}$*)*

25.1 E: Good.
ack$_{30-33}$ *init*$_{34}$*(complete*$_{12}$*)*
init$_{35}$*(begin*$_{16}$*(c-release-objective(*STATE | OBJVS | ID, $\boxed{5}$ | ID*)))*

25.2 E: Loosen the screw in the center and place the jaws around the hub of the wheel…
init$_{36}$*(continue*$_{10}$*(c-identify-recipe(* $\boxed{pull-off}$ | REC | ID, $\boxed{3}$ *[loosen(screw),…])))*
init$_{37}$*(continue*$_{11}$*(c-adopt-recipe(* $\boxed{pull-off}$ | REC | ID, $\boxed{3}$ | ID*)))*