Practical Plug-and-Play Dialogue Management

Danilo Mirkovic

CSLI, Stanford University
Cordura Hall, 210 Panama St, Stanford
CA 94305, USA
danilom@stanford.edu

Lawrence Cavedon

CSLI, Stanford University
Cordura Hall, 210 Panama St, Stanford
CA 94305, USA
lcavedon@stanford.edu

Abstract

We describe an architecture for practical multi-application, multi-device spoken-language dialogue systems, based on the *information-state update* approach. Our system provides representation-neutral core components of a powerful dialogue system, while enabling: scripted domain-specific extensions to routines such as *dialogue move* modeling and reference resolution; easy substitution of specific semantic representations and associated routines; and clean interfaces to external components for language-understanding (i.e. speech-recognition and parsing) and -generation, and to domain-specific knowledge sources. This infrastructure forms the basis of a "plug and play" dialogue management capability, whereby new dialogue-enabled devices can be dynamically introduced to the system. The plug-and-play infrastructure is an important aspect of an environment for dialogue control of in-car devices.

Keywords: multi-device dialogue system; information state; conversational interface.

1 Introduction

CSLI has been developing activity-oriented dialogue systems for a number of years, for applications such as multimodal control of robotic devices (Lemon *et al* 2002), speech-enabled tutoring systems (Clark *et al* 2001), and conversational interaction with in-car devices (Weng *et al* 2004). The dialogue system architecture includes various components: speech-recognizer, language parser, language generation, speech-synthesizer, and the *CSLI Dialogue Manager* (CDM), as well as connections to external application-specific components such as ontologies or knowledge bases, and the dialogue-enabled devices themselves.

Clean interfaces and representation-neutral processes enable the CDM to be used relatively seamlessly with different parsers and language-generation components. Interaction with external devices is mediated by *Activity Models*, i.e. declarative specifications of device capabilities and their relationships to linguistic processes. However, customization to new domains has generally required some significant programming effort, due to variations in dialogue move requirements across applications, as well as certain processes (e.g. reference resolution) having domain-specific aspects to them.

In this paper, we describe recent enhancements to the CDM infrastructure that allows simpler customization to new dialogue domains and applications. Further, this forms the basis of a "plug-and-play" dialogue management architecture: device APIs encapsulate customized dialogue moves, activity models, and knowledge bases, as well as domain-specific extensions to core processes (such as reference resolution). This enables multi-device dialogue management, allowing new dialogue-enabled devices to be dynamically added to an existing multi-device dialogue system.[1]

2 Dialogue Manager Architecture

Figure 1 outlines the CSLI Dialogue System architecture. The CDM has been designed to be used with different components for parsing, NL generation (NLG), etc. Early applications of the CDM used the rule-based head-driven parser GEMINI (Dowding *et al* 1993) with grammars tailored to the particular application domain: the advantage of this approach is that the parser itself performed semantic normalization, returning semantic *logical forms* directly corresponding to the specific representations of device activities. Recent CDM applications have involved use of a third-party statistical parser, returning only weakly normalized semantic forms.

The CDM uses the *information-state update* approach (Larsson and Traum 2000) to maintain dialogue context, which is then used to interpret incoming utterances (including fragments and revisions), resolve NPs, construct salient responses, track issues, etc. Dialogue state is also used to bias speech-recognizer expectation and improve SR performance (Lemon and Gruenstein 2004). Detailed descriptions of the CDM can be found in (Lemon *et al* 2002).

[1] This supplements the plug-and-play approach to dynamic grammars of (Rayner *et al* 2001).

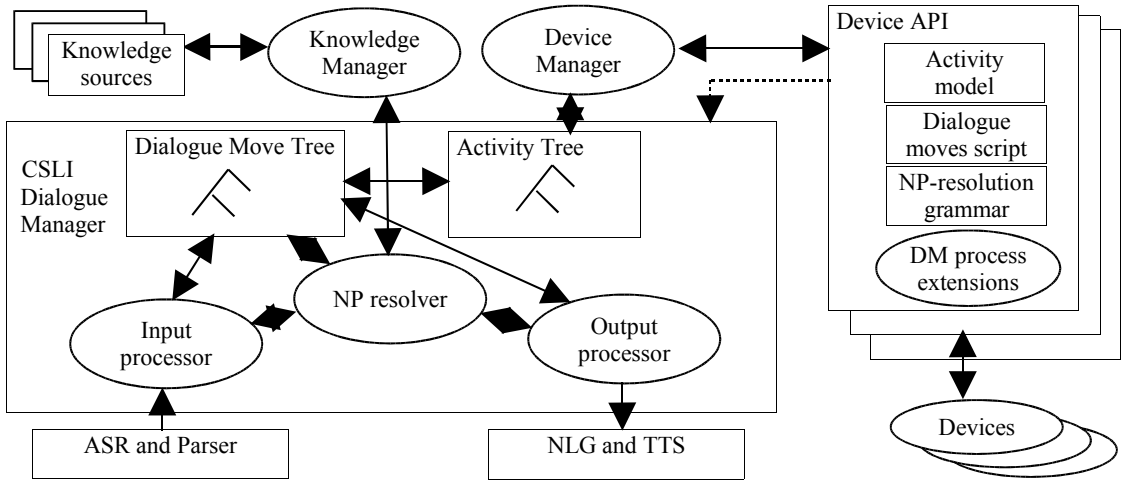

Figure 1: Dialogue System Architecture

The two central CDM components of the dialogue information state are the *Dialogue Move Tree (DMT)* and the *Activity Tree (AT)*. The DMT represents the historical context of a dialogue. Each dialogue contribution is classified as a *dialogue move* (e.g. *Command*, *WhQuestion*, etc.), and is interpreted in context by attaching itself to an appropriate *active* node on the DMT. For example, a *WhAnswer* attaches to an *active* corresponding *WhQuestion* node. The tree structure of the DMT specifically supports multi-threaded, multi-topic conversations (Lemon *et al* 2002): a new conversation topic spawns a new branch. A dialogue move that cannot attach itself to the most recent active node may attach to an active node in another branch---corresponding to a resumed conversation---or open a new branch by attaching itself to the root node---corresponding to a new conversation thread. The DMT also serves as context for interpreting fragments, multi-utterance constructs, and revisions, and provides discourse structure for tasks such as NP-resolution.

The Activity Tree (AT) manages activities relevant to a dialogue. When the user issues a command, this generally results in a new *activity* being created and added to the AT. Before the activity can actually be sent to the device for execution, the system attempts to fully *resolve* it, e.g. resolving all referring NPs or spawning a sub-dialogue to elicit further information. Revisions and corrections (e.g. "I meant/said …") typically involve editing an existing activity representation. Activity-execution is monitored on the AT and changes may result in a notification message being generated, e.g. on failure or successful completion of a task.

Much device-specific information is encapsulated in an *Activity Model*. The Activity Model is a declarative specification of the capabilities of the agent or device with which the CDM interfaces, and includes linguistic information, such as mappings from predicate/argument structure to device-actions. Arguments that are marked as *required* may generate sub-dialogues when a user-command is given with missing arguments. A small portion of an Activity Model for an MP3 device is given in Figure 2. *Playable* in the "required" argument position corresponds to a class from the associated ontology of objects associated with this application; *playable-object* is a variable name filled by matching a dialogue move, as described below.

3 Domain-Independent Dialogue Scripts

In early versions of the CDM, dialogue moves were coded completely programmatically (in Java); our approach involved developing libraries of general-purpose dialogue moves (e.g. *Command*, *WhQuestion*, *WhAnswer*, etc) corresponding to the types of dialogue contributions found in activity-oriented dialogues. As the Dialogue Manager was applied to new applications, new dialogue moves were implemented as necessary, or existing dialogue moves refined to apply to the new application. Multiple applications were implemented in this way.

Customizing dialogue moves to new domains typically required substantial coding. Further, using off-the-shelf parsers with wide-coverage grammars, or corpus-trained statistical parsers, required the CDM to be able to handle new input semantic forms. The requirement of broad coverage dictates that the mapping from input to correct dialogue move be easily extensible.[2] One approach to extending coverage is to normalize semantic information against a broad language ontology (e.g. WordNet (Miller 1995)) or other knowledge base (Dzikovska 2004). However, this still requires incoming forms to be mapped to the internal representation.

[2] Ideally, this could be learned automatically.

To promote re-use of dialogue moves, enhance extensibility, and cope with semantic variation across domains, we have implemented a dialogue-scripting language for writing dialogue moves, defining:

1. hierarchical definition of dialogue moves, allowing inheritance and re-use of existing dialogue moves, while allowing customization to a particular domain or device;
2. direct mappings of input semantic forms to appropriate dialogue moves;
3. attachment rules for information update;
4. other dialogue move-specific information, such as specification of output to be generated for disambiguation, requests for required information, etc.

Note that our approach of using easily-extensible dialogue move scripts is consistent with using other approaches to achieve broad semantic coverage, such as use of an ontology or knowledge-base as mentioned above. However, it provides a general approach for supplying application-specific information to the CDM, for customizing it to new domains, as well as enabling our "plug and play" multi-device infrastructure. Figure 3 illustrates a portion of a sample dialogue move script (details are described below).

```
Types {
    Playable;
    ...
}
Slots {
    Playable playable-object;
    ...
}

// Task definitions
taskdef<play, "play"> {
    DefinableSlots {
           required Playable playable-object;
           optional ...   // optional arguments (e.g. volume)
    }
    ...
}
```

Figure 2: Portion of an Activity Model for an MP3 device

```
User Command:play {
    // inherits from generic Command dialogue move
    Description "play something"
    Input {
              // "play/start X"
           "s( features(mood(imperative)),
               predicate(#play/vb|#start/vb),
                     ?arglist(obj:_,?sbj:*))"
             // "I want to play/hear X"
           "s( features(mood(indirect)),
               predicate(#play/vb|#hear/vb),
                     ?arglist(obj:_playable-object,?sbj:*))"
           // other templates ...
    }
    Producing {     // Questions
           System WHQuestion:disambiguate
           System WHQuestion:fill:play:_playable-object {
                 Output {avs "(e1 / play
                                  :question (q1 / what)
                                  :agent I)"
                 } ... }
                 CloseOn System Report:play:playing {
                       Output {avs "(e1 / play
                                   :patient (p1 / [song])
                                   :aspect continuous)"
                 }
           ... }
... }
```

Figure 3: Sample dialogue move script for a *play* Command for an MP3 device

Variables in the dialogue move script correspond to variables in the Activity Model (AM) for the corresponding device. In particular, the AM for the MP3 device contains a *play* operation with a corresponding (required) *_playable-object* argument. When an incoming semantic form matches an *Input* template in the above script, the unification operation fills the *_playable-object* variable, which resolves to an object from the device's domain of objects (see below) and fills the corresponding slot in the activity constructed from the device Activity Model.

Following are further details on the properties of the dialogue move scripting language.

3.1 Hierarchical dialogue move specification

The scripting language allows hierarchical specification and refinement of dialogue moves. The move in Figure 3 corresponds to a *play* command, and inherits from a more generic *Command* dialogue move. The *Command* dialogue move is implemented in Java: its script has a field naming the Java class that implements it. The *play* command move is implemented by the same generic code, but specifies its own patterns for triggering the move, and defines attachment patterns and appropriate generation messages. In general, the depth of inheritance is unbounded: e.g. we could define a sub-move of the *play* command move that is applicable in very specific contexts.

One type of move for which this is particularly useful is information-query moves across devices. Questions about music in an MP3 database or restaurants in a city information guide are often structurally similar: i.e. query-construction itself is (relatively) domain-independent. Each type of query can be handled by a different dialogue move (corresponding to different devices or knowledge sources), but each set of *Inputs* is inherited from a single *Query* dialogue move.

Other operations that can be applied at abstract levels of dialogue move include *rewrite rules*: these are used to transform input forms before they are matched against dialogue move *Input* templates, e.g., transforming indirect commands into direct imperatives, or replacing a temporal marker (e.g. "now") with an appropriate semantic feature. Rewrite rules are attached to domain-specific dialogue move scripts, or to generic scripts, as appropriate.

3.2 Selecting dialogue move via semantic template

The *Input* section of a dialogue move script contains the list of input items that would trigger this particular dialogue move. These templates are matched against the output of the parser (in the case of the example in Figure 3, a statistical parser trained on a corpus collected from Wizard of Oz experiments of users interacting with an MP3 player (Cheng *et al* 2004)). Parsed forms may be normalized or processed in any way (e.g., using an ontology, or via rewrite rules) before being matched against *Input* templates. *Input* templates can be attached to domain-specific dialogue moves or to generic moves (and inherited).

The specific formalism of the *Input* templates in Figure 3 is not important: the formalism is specific to the output of a particular statistical parser, but the templates can be viewed as feature structures, and the matching operation is effectively one-way unification. The special symbols are interpreted as follows: "#" indicates a lexical item, with part-of-speech tag following the "/" symbol; "|" indicates alternatives; "?" indicates an optional argument; "_" indicates a variable matching one from the Activity Model; and "*" matches anything. Hence, the dialogue move in Figure 3 matches "play X", "start X", or an indirect command involving "play X" or "hear X";[3] the object to be played is marked as optional---i.e., the template matches even when this argument is missing.

The CDM is representation neutral, in that the form of the templates and the corresponding matching algorithm can be replaced without affecting the rest of the CDM infrastructure. This enables easy replacement of parser or NLG component to one using different representations. For example, we could substitute a more standard feature-structure representation and feature-unification algorithm, with no other changes to the Dialogue Manager code required.

When an input form matches an entry in a dialogue move's *Input* section, this may cause variables to be bound; in particular, a variable that corresponds to one from the Activity Model. For example, if an input matching the dialogue move in Figure 3 contains a well-formed *arg* argument, then this supplies a value for *_playable-object*; if no *arg* is present, then this variable is left unfilled (in which case the *Command* dialogue move generates a request for information).

In general, multiple matches are possible, since there are generally multiple scripted dialogue moves and multiple entries in each move's *Input* section. Our current strategy is to score each possible match using generic criteria (e.g. applicability to current context; minimizing unresolved information). Current work inolves investigating probabilistic approaches to incorporating evidence from multiple criteria to select appropriate dialogue move, including prosodic information and shallow topic-categorization.

3.3 Specifying attachment rules

The dialogue scripting language provides a mechanism for specifying attachment rules: i.e. which types of dialogue moves can attach to an existing active node in the DMT. For example, Figure 3 shows that (amongst others) a disambiguating *WhQuesion* or a *WhQuestion* for filling a missing argument can attach to a *Command*

[3] For convenience, indirect commands have their embedded sentence extracted using a generic rewrite rule.

node.[4] Dialogue move information can be scripted "in place" inside one of these specifications (as done for the *WhQuestion:fill:play* move).
The scripts also encode which adjacent moves close a dialogue move (i.e. inactivate it so no other move can attach to it), in the *CloseOn* field. Closing a node for attachment effectively closes the corresponding thread of conversation.[5] Nodes are also automatically closed after a specified period.

3.4 Specification of system responses

Much of the system output is automatically generated, e.g. encoded in very general-purpose dialogue moves. However, applications often call for domain- and device-specific outputs. These can also be encoded in the dialogue move scripts; since these will be system responses, these are encoded inside *System* dialogue moves. Any representation is permitted, so long as it matches the representation used by the specific NLG system with which the CDM is interfaced for the given application. This is another way in which the CDM is representation neutral: including a different NLG component, using a different representation, in the dialogue system may require modification of the scripts, but requires no modification to the core of the CDM.

4 Multi-Device Plug-and-Play Dialogue Management

Plug-and-play capability is an important feature of systems that can have their functionality extended without going off-line. Plug-and-play typically involves adding new components that provide enhanced functionality without disrupting the existing framework. Implementing a plug-and-play environment requires at minimum a specification language for components to advertise their capabilities, as well as a clean encapsulation of the implementation of the component. In our framework, the first of these is provided as part of the dialogue-move scripting; we discuss other aspects of device-encapsulation here.

4.1 Device encapsulation

New devices that register with the dialogue manager must encapsulate all information required for managing dialogue with these new devices. This information includes:

1. dialogue-move scripts, as described above;
2. Activity Model describing any device functionality accessible by dialogue;
3. device-specific ontology and knowledge base (KB);
4. rules for device-specific NP-resolution (Section 4.3).

Device-specific implementations of dialogue management processes can also be added, or overwrite generic ones, by the device encapsulation including appropriate new Java classes. For example, a dialogue-move that handles a new form of interaction introduced by a new device could be added. In general, however, the four components above contain all device-specific information required, and allow for dynamic plug-and-play of dialogue-enabled devices.

4.2 Multi-device dialogue management

We have extended the Dialogue Move Tree (DMT) infrastructure so that it allows new devices to be plugged in dynamically. New dialogue-enabled devices register themselves with a Device Manager. Nodes in the DMT are associated with specific devices where appropriate; "current device" becomes part of the information-state and interpreting incoming utterances is performed in this context.
Device-selection---i.e., determining which device an utterance is associated with---is a further complication in this setting. The current decision process involves lexical and semantic information, dialogue move classification, discourse structure, as well as bias towards the "current device". Relating NPs to the appropriate device ontology is a specific strategy: e.g. reference to a "song" will match a category in the ontology associated with an MP3 device, but potentially with no other devices. This strategy does not necessarily resolve all device-ambiguities however: e.g. an address-book may be used by both a phone-device ("get John on the phone") as well as a navigation service ("how do I get to John's house?"). In general, the processes of device-selection and NP-resolution are co-dependent: information about the resolution of NPs provides important clues about the device being referred to; however, NP-resolution may actually be quite device-specific (as discussed in Section 4.3). Our approach is to perform a shallow NP analysis, e.g. matching nouns and proper names against ontology categories and KB items associated with a specific device in order to identify device, and then using the device-specific NP-resolution rules (see Section 4.3) to fully resolve the NPs.
We plan to use other features (e.g., shallow topic-categorization techniques) and develop probabilistic methods for this classification task. We are currently collecting a corpus of multi-device dialogues and also hope to be able to automatically learn this task.

4.3 Multi-device NP-resolution

Much of the NP-resolution process can be seen as fairly domain-independent (e.g. anaphora resolution). However, aspects of NP-resolution are both domain- and device-dependent. For example, interpreting "What's this"

[4] Such attachment rules are often specified at more abstract levels, not at the level of specific commands.

[5] Revisions may reopen a *Command* or *Query* node.

is handled differently in the context of music playing over an MP3 player than when using a touch-screen multimodal interface.

We address this problem in a manner analogous to our approach to customizing dialogue moves: the core NP-resolution capabilities implemented in the CDM can be custom-adapted for a specific domain/device via an NP-resolution scripting language. NP-resolution scripts are effectively context-free grammars that allow the user to define how NP objects are mapped to knowledge-base queries for a specific device, in the context of the current dialogue information state and input semantic form. In particular, for the MP3 device, "this" in the context of "What's this" would be mapped to a query that returns the name of the currently playing song.

NPs are translated into Java objects implementing constraint-based KB-queries. Rules specify how to translate NPs specified in the input semantic form into such objects. The dialogue manager contains a number of generic constraint objects and associated transformations, but further objects may be included as part of the device encapsulation to provide any novel processing specific to that device. For example, an MP3 device will need to handle qualifiers such as "by *artist*", and know to translate this construct into an appropriate constraint on the *artist* field of the KB.

The way in which information is extracted from an NP representation depends, of course, on the specific format of the input as well as the structure of the KB associated with a device. We use a rule-based language for specifying how an NP (in whatever format is used) maps to constraint-based query objects, making use of generic or device-specific frame-construction operations. Such rules are used for handling synonyms (i.e. by mapping multiple nouns into the same query-type) as well as specifying the construction of complex query objects from complex NPs.

The simple examples in Figure 5, taken from the NP-resoltion script for an MP3 device, illustrate some of the features of our approach. The left-hand side of each rule matches a construct from the output of the statistical parser used in this particular application: the symbol '#' indicates that the corresponding word has been marked as a head word; the token following '/' is the POS of the matched item; entries in upper-case designate variables. The right-hand side of each rule specifies how to construct a query or constraint for the KB: the first item signifies what type of constraint this is (which determines which construction process to use); the rest of the RHS specifies the specific KB fields to test.

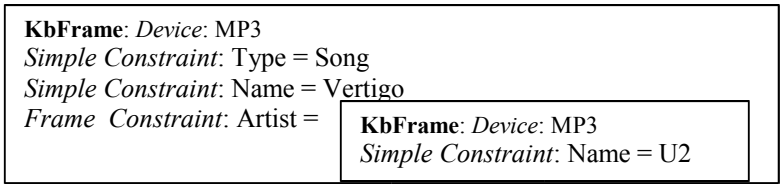

Figure 4: Sample NP-query object for "the song Vertigo by U2"

1. #song --> *Simple* system:hasCategory : music:Song
   "What *songs* do you have?"
2. #this/dt --> *MP3Specific* this music:Song
   "What's *this*?"
3. ncomp(#by/in,sbj:FRAME) --> *Frame* music:songHasArtist|music:albumHasArtist
   "Do you have anything *by* X?"
4. s(predicate(#called|#named),arglist(obj(#:WORD))) --> *Simple* system:hasName
   "Do you have a song *called/named* X?"
5. ncomp(#on/in|#in/in|#from/in,sbj:FRAME) --> *Frame* music:hasSongList
   "Play something *from* an album by X"

Figure 5: Example NP-resolution rules

In the figure, (1) constructs a simple query for objects of type "Song"; (2) utilizes an MP3-specific constraint-construction processes to match a "Song" labeled as "this" (which is mapped to the currently playing song);[6] (3) maps a "by X" complement onto a constraint on "Artist" (for either a song or an album)---the FRAME variable indicates that the embedded NP may be complex and should be itself analyzed; (4) handles an embedded sentence of the form "named X"; and (5) handles other complements that may qualify a query for a song.

[6] Note that this only matches when "this" is used on its own, not as a determiner in an NP, as in "this artist"---that case is handled separately.

The interesting issue here is not so much the particular syntax or format of our rules but the fact that such constructions can be easily specified for a given new device and encapsulated with the device. As with dialogue-move scripts, generic constructs can be inherited or overwritten in a device-specific script, while device-specific NP-interpretation rules can be encapsulated with the given device. This is an important aspect of the plug-and-play device specification.

5 Related Work

Multi-domain dialogue management typically involves taking a powerful generic dialogue management engine (e.g. (Allen *et al* 2000)) and customizing it for a new domain by providing domain-specific information such as language grammars and knowledge-bases. Extending such systems to be multi-device applications usually involves extending domain-specific components and processes so as to cover the new application domains. Dzikovska *et al* (2003) describe a novel way of using two ontologies---one associated with a broad-coverage parser, and a second associated with a task-domain---to reuse as much generic linguistic capabilities when customizing their dialogue system to new domains. Such approaches are consistent with the approach we have taken: they provide a canonical semantic form for input to the CDM, with the dialogue-move scripting approach being used to deal with any missing coverage.

Dynamic plug and play architectures for devices and services are becoming widespread in industry, e.g. via interface standards for Web services and the *service-oriented computing* paradigm (Singh and Huhns 2005). Dynamic management of intelligent devices has also long been an important topic of multi-agent systems research (e.g. the Open Agent Architecture (Cheyer and Martin 2001)).

For dialogue systems, (Rayner *et al* 2001) supplement device plug-and-play with device-specific speech and language processing capabilities. In particular, adding a new device to the system can extend the generic unification grammar for speech and language processing, by adding one or more of: new lexical entries, new grammar rules, or new feature values. Rayner *et al*'s techniques are specifically designed to allow grammars to be built by incremental addition while using existing and newly introduced features to constrain rule application, to ensure maximally tight speech-recognition language models. Semantic interpretation rules can also be included with a device. However, task descriptions and other dialogue phenomena are not the main focus of their work and these representation processes are simple. Rayner *et al*'s work is set in the context of an intelligent home, where new dialogue-enabled devices are dynamically added to the system.

The problem of device-resolution within this same application area is addressed in (Quesada and Amores 2002). Device-properties (e.g. *location*, *device-type*) are hierarchically-structured, allowing appropriate device in a referring expression to be resolved by lookup (e.g. for "outdoor lights", *location=outdoors* and *device-type= light*). Dynamic addition of devices involves categorizing such new devices appropriately in the device KB. However, this work does not address how other dialogue processes are dynamically extended on adding a device.

The approach to plug-and-play dialogue management taken by Pakucs (2002, 2003) has similarities to that taken here. In particular, Packucs' SesaME system associates a *Dialogue Description Collection (DDC)* with the dialogue engine; these are associated with services or devices and are updated on changes to these. DDCs contain simple dialogue scripts, written in a formalism based on VoiceXML, and contain task- and domain-specific dialogue scripting, as well as any application-independent scripting for error-handling and meta-dialogues (e.g. for providing information about available services). In essence this is similar to the general approach we have taken, although within a much lsimpler dialogue-management framework.

Many dialogue-management systems support interaction with multiple devices, and may use a device-manager component to support dynamic integration of new devices. Such approaches typically involve extending data and processes to be broader-coverage so as to cover extended domains introduced by these new devices, as opposed to encapsulating the extensions to the dialogue manager with the dialogue-enabled device.

Our approach also contrasts with other implementations based on the TrindiKit (Larsson and Traum 2000), such as DIPPER (Bos *et al* 2003), which provides a rich collection of primitive constructs for building a dialogue manager. Rather, we provide a powerful practical implementation of a core system, customizable via scripting to new domains and applications, paying particular attention to encapsulating device information so as to enable a plug-and-play dialogue management system.

6 Discussion

We have presented an extension to the CSLI Dialogue Manager that enhances extensibility, customization, and reuse, as well as forming the basis of a multi-device plug-and-play dialogue system. Our approach contrasts with other implementations based on the TrindiKit (Larsson and Traum 2000), such as DIPPER (Bos *et al* 2003), which provides a rich collection of primitive constructs for building a dialogue manager. Rather, we provide a powerful practical implementation of a core system, customizable via scripting to new domains and applications, paying particular attention to encapsulating device information so as to enable a plug-and-play dialogue management system.

The CDM and the plug-and-play architecture is part of a system for dialogue control of in-car electronic components, such as entertainment systems, navigation system, and telematic devices.[7] The vision is a framework whereby new devices, or dialogue-capability for existing devices, can be added easily and without disruption to the existing infrastructure.

The device-encapsulation approach, and in particular the dialogue move scripting language and NP-resolution rules described here, has been applied to an initial domain---controlling an MP3 music player and accessing a music database. Evaluation of a multi-device system, involving the MP3 player and a restaurant recommendation capable of also providing navigational directions, is scheduled for July 2005; evaluation results will be available for presentation at the workshop.

Acknowledgement. This work is supported in part by the NIST Advanced Technology Program.

References

J. Bos, E. Klein, O. Lemon, T. Oka. 2003. Dipper: Description and formalization of an information-state update dialogue system architecture. *4th SIGdial Workshop on Discourse and Dialogue*, Sapporo.

H. Cheng, H. Bratt, R. Mishra, E. Shriberg, S. Upson, J. Chen, F. Weng, S. Peters, L. Cavedon, J. Niekrasz. 2004. A Wizard of Oz framework for collecting spoken human-computer dialogs. *INTERSPEECH: 8th International Conference on Spoken Language Processing*, Jeju Island, Korea.

A. Cheyer and D. Martin. 2001. The Open Agent Architecture. *Journal of Autonomous Agents and Multi-Agent Systems*, 4(1).

B. J. Clark, Fry, M. Ginzton, S. Peters, H. Pon-Barry, Z. Thomsen-Grey. 2001. Automated tutoring dialogues for training in shipboard damage control. *2nd SIGdial Workshop on Discourse and Dialogue,* Aalborg.

J. Dowding, J. M. Gawron, D. Appelt, J. Bear, L. Cherny, R. Moore, D. Moran. 1993. GEMINI: A natural language system for spoken-language understanding. *31st Meeting of the ACL*, Columbus, OH.

M. O. Dzikovska. 2004. *A Practical Semantic Representation for Natural Language Parsing*. Ph.D. Thesis, University of Rochester.

M. O. Dzikovska, M. D. Swift, J. F. Allen. 2003. Integrating linguistic and domain knowledge for spoken dialogue systems in multiple domains. *IJCAI-03 Workshop on Knowledge and Reasoning in Practical Dialogue Systems*, Acapulco, Mexico.

S. Larsson and D. Traum. 2000. Information state and dialogue management in the TRINDI dialogue move engine toolkit. *Natural Language Engineering,* 6(3-4).

O. Lemon and A. Gruenstein. 2004. Multi-threaded contect for robust conversational interfaces: context-sensitive speech-recognition and interpretation of corrective fragments. *Transactions on Computer-Human Interaction (ACM TOCHI)*, 11(3).

O. Lemon, A. Gruenstein, S. Peters. 2002. Collaborative activities and multi-tasking in dialogue systems. *Traitement Automatique des Langues (TAL),* 43(2).

G. A. Miller. 1995. WordNet: A lexical database for English. *Communications of the ACM 38.*

B. Pakucs. 2002. VoiceXML-based dynamic plug and play dialogue management for mobileenvironments. *ISCA T&R Workshop on Multi-Modal Dialogue in Mobile Environments,* Kloster Irsee, Germany.

B. Pakucs. 2003. Towards dynamic multi-domain dialogue processing. *EUROSPEECH: 8th European Conference on Speech Communication and Technology*, Geneva.

J. F. Quesada and J. G. Amores. 2002. Knowledge-based reference resolution for dialogue management in a home domain environment. *EDILOG: 6th Workshop on the Semantics and Pragmatics of Dialogue*, Edinburgh.

M. Rayner, I. Lewin, G. Gorrell, J. Boyce. 2001. Plug and play speech understanding. *2nd SIGdial Workshop on Discourse and Dialogue*, Aalborg.

M. P. Singh and M. N. Huhns. 2005. *Service-Oriented Computing: Semantics, Processes, Agents*. John Wiley and Sons.

F. Weng, L. Cavedon, B. Raghunathan, D. Mirkovic, H. Cheng, H. Schmidt, H, Bratt, R. Mishra, S. Peters, L. Zhao, S. Upson, L. Shriberg, C. Bergmann. 2004. A conversational dialogue system for cognitively overloaded users (poster). *INTERSPEECH: 8th International Conference on Spoken Language Processing*, Jeju Island, Korea.

[7]Collaborating partners are Robert Bosch Corporation, VW America, and SRI International.