Plug and Play Speech Understanding

Manny Rayner, Ian Lewin & Genevieve Gorrell
netdecisions Ltd
Wellington House,
East Road, Cambridge CB1 1BH, UK
manny.rayner|ian.lewin|genevieve.gorrell
@netdecisions.com

Johan Boye
Telia Research
S-123 86 Farsta, Sweden
johan.boye@trab.se

Abstract

Plug and Play is an increasingly important concept in system and network architectures. We introduce and describe a spoken language dialogue system architecture which supports Plug and Playable networks of objects in its domain. Each device in the network carries the linguistic and dialogue management information which is pertinent to it and uploads it dynamically to the relevant language processing components in the spoken language interface. We describe the current state of our plug and play demonstrator and discuss theoretical issues that arise from our work. Plug and Play forms a central topic for the DHomme project.

1 Introduction

The notion of Plug and Play finds its most natural home in the world of networked home devices, where it offers at least the following two important properties

- the network of devices is dynamically reconfigurable as devices are brought online or disappear offline
- zero re-configuration by the user is required

Frameworks for achieving Plug and Play generally address this by including at least the following

- devices announce themselves on the network when they are plugged into it (and also discover the existence of others)
- devices describe their own capabilities, provide a means for accessing them and can query and access the capabilities of others
- devices should support, where possible, seamless interaction with other devices.

Plug and Play is, not surprisingly, viewed as a pre-requisite for the commercial success of networked devices in the home. There are already several promising candidate platforms for achieving the necessary functionality, including Universal Plug and Play (UPnP) (Microsoft, 2000) and Jini (Oaks and Wong, 2000). In this paper, we address the requirements on spoken dialogue interfaces that arise from a plug and play domain. We also present the current state of our English language plug and play demonstrator for controlling lamps, dimmers and sensors, previously described in (Rayner et al., 2001b). (There is also a Swedish instantiation).

First, however, we need briefly to distinguish our notion from other notions of plug and play and reconfigurability.

The notion of Plug and Play has been used for dialogue system toolkits in which the various different language processing components themselves (e.g. recognition, parsing, generation and dialogue management) can be plugged in and out. The most prominent instance of this is the Darpa Communicator architecture (Goldschen and Loehr, 1999), which defines interoperability standards for language processing components. The intention is simply that researchers and developers can experiment with systems containing different instantiations of the language processing components. The Communicator Architecture is not designed to address the special requirements of a plug and play domain. In fact, the Communicator architecture does not support the dynamic re-configuration of language processing components while the system is running.

At a more general level, simple re-configuration of spoken language dialogue systems has of course long been a goal of language engineering. But such re-configuration is nearly always viewed as the problem of *cross-domain* or possibly *cross-language* porting, e.g. (Glass, 1999). Once one has a cinema ticket booking service, for example, one may examine the effort required for booking train tickets, or for e-shopping in general or even the "database access" scenario. There are various toolkits, architectures and methodologies for rapidly and/or semi-expertly generating new instances of dialogue systems, e.g. by abstracting away from domain or application dependent features of particular systems, e.g. (Fraser and Thornton, 1995; Kolzer, 1999), or 'bottom-up' by aggregation of useful re-configurable components , e.g. (Sutton et al, 1998; Larsson and Traum, 2000). The automated within-domain reconfiguration required for a plug and play domain, has not, to our knowledge, been described previously.

Pursuit of plug and play functionality (and its realization in strong and weak forms - discussed in section 3) forms a central theme of the DHomme project. [1]

In the rest of this paper, we begin by detailing our concrete Plug and Play scenario - device control in the home - with an example dialogue from our demonstrator and an outline of the main dialogue processing elements. In section 3, we distinguish strong and weak notions of Plug and Play and their applicability to spoken language interfaces. In section 4, we discuss the strong plug and play capability we have built into the recognition, parsing and (context independent) semantic interpretation system components of our demonstrator. In section 5, we discuss some future work for Plug and Play dialogue management. Section 6 contains our conclusions.

2 A Plug and Play Scenario

In this section we present example dialogues from our current demonstrator and briefly outline the main processing elements. The domain is networked home devices, an area where plug and play is already in a reasonably advanced state of development and speech control looks highly attractive.

2.1 Plug and Play Examples

Figure 1 displays an example dialogue from our current demonstrator.

In U1, the user asks for the TV to be switched on. The system reports (S1) that it simply does not understand the user. (If it possessed "TV" in its recognition vocabulary and knew something about the concept of TVs it could have reported *I don't know of any TV*). When a TV is plugged into the network (following U2), the system is able to understand, and act on the user's repeated request to switch on the TV. The system reports on its action (S3). S4 illustrates another type of "error" message. When another TV is plugged into the network, the system must now engage in disambiguation behaviour whereas previously it had no need to (S7).

S10 illustrates that, in the absence of dimmable lights, "Dim" is not understood, and, possibly, not even recognized. When a dimmable light is plugged in (or, at least, knowledge of dimmable lights is plugged in), then a more helpful error message can be given in S12. Finally, when the grammar is increased to cover new commands, the system may begin to make mistakes that it did not make originally (S13).

2.2 The current demonstrator

Our demonstrator expects devices of three main types: switchable, dimmable and sensors. Switchable devices are binary state devices that can be set or queried. Dimmable devices have a state varying on a single scalar dimension which can be set, changed, or queried. Sensors are similar but can only be queried.

Formally, these commands and queries are encoded by a 4-ary slot-value structure and Device Grammars must generate semantic values containing these slots. (Not all slots are required for every utterance, of course.) The four slots are: `op` (filled by *command* or *query*); `level`; `change` and `dir` (*on* or *off*). In order to identify devices, there are 4 other slots: `device` (*light*, *tv* ...), `loc` (*bathroom*, *kitchen* ...), `device-spec` (*all*, *the* ...) and `pronoun` (*it*, *them* ...). For example, *Is the light in the hall on?* translates to ⟨ `op`=*query* `dir`=*on* `device`=*light* `device-spec`=*the* `loc`=*hall* ⟩. *Dim everything by ten percent* translates to ⟨ `op`=*command* `dir`=*off* `device-spec`=*everything* `change`=*10* ⟩. *Switch the hall and kitchen lights off* translates to ⟨ `op`=*command* `dir`=*off* `level`=*0* `device`=*light* ⟨ `loc`=*kitchen* `loc`=*hall* ⟩⟩.

Dialogue interpretation contains three stages. First, conjunctions in the input (which are treated as just packed representations) are unpacked into a set of (7-ary) slot-structures. Secondly, a form of ellipsis resolution, "sticky defaults", takes place in which missing slots are filled in from their previous values. A fragmentary semantic value is simply pasted over the corresponding parts of the last one. Thus *And in the bathroom* translates to ⟨ `loc`=*bathroom*⟩ but the other required slots (e.g. `device`) are supplied from the previous representation. Finally reference resolution tries to determine device identifiers for pronouns and definitely described devices. Currently, devices are identified only by their location and type so a simple

[1]This work is supported by EU 5th Framework project IST-2000-26280 – see Acknowledgments

matching procedure can be used.

Following contextual interpretation, either a program (a command or sequence of such) or an 'error' condition (e.g. resolution failed to identify a device) will have been generated. The system must then execute the program and/or generate feedback. Knowledge of how to execute these programs, e.g. that it is an 'error' to try to switch on a light that is already on, and possible feedback messages are simply hardcoded into the Dialogue Manager. There is a pre-defined set of feedback message structures associated with each underlying action and their possible results. Some example paraphrases of message structures are "The X is now on", "The X is already on", "The X is now at Y percent", "There is no X in the Y".

3 Strong and Weak Plug and Play

In its weakest form, Plug and Play refers only to the ability to add a device to a network without manual configuration. Knowledge distribution is not included. Standard Plug and Play for PC peripherals simply automates the matching up of physical devices with software device-specific drivers in the PC. Communication links between them are established by reserving resources such as shared memory and interrupt request numbers. The weak sense is very useful. Users need not configure their hardware via jumper switches or software drivers by entering 'magic' numbers in configuration files.

In the strong sense, Plug and Play can refer also to modular, distributed knowledge. Devices not only set up network communications but publish information about themselves over it. Other devices can obtain and use it. In Jini, for example, a new printer device can register its printing service (and java code for invoking methods on it) on the network. Then, a word-processing application can find it and configure itself to use it. In UPnP, devices publish XML descriptions of their interfaces.

The strong-weak contrast is not a sharp or binary one. The word-processor might know the industry agreed printer interface and so display a greyed-out print button if no printer is networked. When a new type of printer is networked, it might supply additional print options (e.g. "print colour") that the processor knows nothing about.

One desirable Plug and Play property in both strong and weak forms is commutativity, i.e. the system understands the same commands in the same way no matter which device is connected first. It is less obvious whether disconnecting device X should be the inverse operation of connecting device X. This seems reasonable in a weak plug and play system, but in the strong case it would mean that the recognizer would cease to understand the word "TV" as soon as the TV were disconnected. This might be confusing for the user.

The strong and weak senses of plug and play apply to spoken language dialogue interfaces. In the weakest sense, the dialogue system might be entirely pre-configured to deal with all possible devices and device-combinations. The required knowledge is already present in the network. Plug and Play then consists of identifying which particular devices are currently networked and establishing communication channels with them. In the stronger sense, the components of the spoken language dialogue interface acquire the knowledge pertinent to particular devices from those devices. So, as in example S1 above, the speech recognizer may not have the word "TV" in its vocabulary until a TV is plugged into the network. The dialogue manager may not be capable of uttering "That device is not dimmable" until a dimmable device is plugged into the network. A strongly Plug and Play system may therefore be distinguishable from a weaker one by its behaviour in the absence of certain device specific knowledge. If the relevant knowledge is present, one cannot be certain whether it was pre-configured or uploaded "on demand".

Plug and Play also enforces a certain sort of modularity on the system. Since devices must declare the information required to update the dialogue components, a clear interface is provided for re-configuring the system for new types of device as well as a clearer picture of the internal structure of those dialogue components. Indeed, it is really just a design choice whether device knowledge is in fact installed only when the device is plugged in. One may, for example, choose to optimize recognition performance on the set of devices actually installed by not loading information about other devices. Alternatively, one might prefer to recognize the names of devices not installed so that helpful error messages can be delivered.

Potentially, each component in a spoken language interface (recognizer, parser, interpreter, dialogue manager etc.) can be updated by information from a device in a Plug and Play domain. Different components might support different degrees of strength of the Plug and Play notion. Furthermore, different instantiations of these components may require very different sorts of update. To take a very simple example, if recognition is carried out by a statistically trained language model, then updating this with information pertinent to a particular device will evidently be a significantly different task from updating a recognizer which uses a grammar-based language model.

Our current demonstrator program instantiates a Plug and Play capability for recognition, parsing and context independent semantic analysis and is built on top of the Nuance toolkit (Nuance Communications, 1999). The next section discusses the capability in detail. Section 5 discusses and makes some proposals for Plug and Play Dialogue Management.

4 Distributed Grammar

4.1 Introduction

In this section, we will describe how we have addressed the issues that arise when we attempt to apply the (strong) Plug and Play scenario to the tasks of speech recognition and language processing. Each device will provide the knowledge that the speech interface needs in order to recognise the new types of utterance relevant to the device in question, and convert these utterances into well-formed semantic representations.

Let's start by considering what this means in practice. There are in fact a whole range of possible scenarios to consider, depending on how the speech understanding module is configured. If the module's construction is simple enough, there may be no significant problems involved in extending it to offer Plug and Play functionality. For example, the command vocabulary offered by the speech interface may just consist of a list of fixed phrases. In this case, Plug and Play speech recognition becomes trivial: each device contributes the phrases it needs, after which they can be combined into a single grammar. An approach of this kind fails however to scale up to an interface which supports complex commands, in particular commands which combine within the same utterance language referring to two or more different devices. For example, a command may address several devices at once ("turn on the radio and the living room light"); alternately, several commands may be combined into a single utterance ("switch on the cooker and switch off the microwave"). Our experience with practical spoken device interfaces suggests that examples like these are by no means uncommon.

Another architecture relatively easy to combine with Plug and Play is doing recognition through a general large-vocabulary recogniser, and language-processing through device-specific phrase-spotting (Milward, 2000). The recogniser stays the same irrespective of how many devices are connected, so there are by definition no problems at the level of speech recognition, and it is in principle possible to support complex commands. The main drawback, however, is that recognition quality is markedly inferior compared to a system in which recognition coverage is limited to the domain defined by the current set of devices.

Modern speech interfaces supporting complex commands are typically specified using a rule-based grammar formalism defined by a platform like Nuance (Nuance Communications, 1999) or SpeechWorks (Inc, 2001). The type of grammar supported is some subset of full CFG, extended to include semantic annotations. Grammar rules define the language model that constrains the recognition process, tuning it to the domain in order to achieve high performance. (They also supply the semantic rules that define the output representation; we will return to this point later). If we want to implement an ambitious Plug and Play speech recognition module within this kind of framework, we have two top-level goals. On the one hand, we want to achieve high-quality speech recognition. At the same time, standard software engineering considerations suggest that we want to minimize the overlap between the rule-sets contributed by each device: ideally, the device will only upload the specific lexical items relevant to it.

It turns out that our software engineering objectives conflict to some extent with our initial goal of achieving high-quality speech recognition. Consider a straightforward solution, in which the grammatical information contributed by each device consists purely of lexical entries, i.e. entries of the form

```
<Nonterminal> --> <Terminal>
```

In a CFG-based framework, this implies that we have a central device-independent CFG grammar, which defines the other rules which link together the nonterminals that appear on the left-hand-sides of the lexical rules. The crucial question is what these lexical non-terminal symbols will be. Suppose, for concreteness, that we want our set of devices to include lights with dimmer switches, which will among other things accept commands like "dim the light". We might achieve this by making the device upload lexical rules of the rough form

```
TRANSITIVE_VERB --> dim
NOUN --> light
```

where the LHSs are conventional grammatical categories. (We will for the moment skip over the question of how to represent semantics). The lexical rules might combine with general grammar rules of the form

```
COMMAND --> TRANSITIVE_VERB NP
NP --> DET NOUN
DET --> the
```

This kind of solution is easy to understand, but experience shows that it leads to poor speech recognition. The problem is that the language model produced by the grammar is underconstrained: it will in particular allow any transitive verb to combine with any NP. However, a verb like "dim" will only combine with a restricted range of possible NPs, and ideally we would like to capture this fact. What we really want to do is parameterise the language model. In the present case, we want to parameterise the TRANSITIVE_VERB "dim" with the information that it only combines with object NPs that can be used to refer to dimmable devices. We will parameterise the NP and NOUN non-terminals similarly. The obvious way to do this within the bounds of CFG is to specialise the rules approximately as follows:

```
COMMAND --> TRANS_DIM_VERB DIMMABLE_NP
DIMMABLE_NP --> DET DIMMABLE_NOUN
TRANS_DIM_VERB --> dim
DIMMABLE_NOUN --> light
DET --> the
```

Unfortunately, however, this defeats the original object of the exercise, since the "general" rules now make reference to the device-specific concept of dimming. What we want instead is a more generic treatment, like the following:

```
COMMAND -->
   TRANSITIVE_VERB:[sem_obj_type=T]
   NP:[sem_type=T]
NP:[sem_type=T] -->
   DET NOUN:[sem_type=T]

DET --> the
TRANSITIVE_VERB:[sem_obj_type=dimmable]
   --> dim
NOUN:[sem_type=dimmable] --> light
```

This kind of parameterisation of a CFG is not in any way new: it is simply unification grammar (Pullum and Gazdar, 1982; Gazdar et al., 1985). Thus our first main idea is to raise the level of abstraction, formulating the device grammar at the level of unification grammars, and compiling these down into the underlying CFG representation. There are now a number of systems which can perform this type of compilation (Moore, 1998; Kiefer and Krieger, 2000); the basic methods we use in our system are described in detail elsewhere (Rayner et al., 2001a). Here, we focus on the aspects that are required for "distributed" unification grammars needed for Plug and Play.

4.2 "Unification grammars meet object-oriented programming".

Our basic idea is to start with a general device-independent unification grammar, which implements the core grammar rules. In our prototype, there are 34 core rules. Typical examples are the NP conjunction and PP modifications rules, schematically

```
NP --> NP CONJ NP
NP --> NP PP
```

which are likely to occur in connection with any kind of device. These rules are parameterised by various features. For example, the set of features associated with the NP category includes grammatical number (singular or plural), WH (plus or minus) and sortal type (multiple options).

Each individual type of device can extend the core grammar in one of three possible ways:

New lexical entries A device may add lexical entries for device-specific words and phrases; e.g., a device will generally contribute at least one noun used to refer to it.

New grammar rules A device may add device-specific rules; e.g., a dimmer switch may include rules for dimming and brightening, like "another X percent" or "a bit brighter".

New feature values Least obviously, a device may extend the range of values that a grammatical feature can take (see further below).

For usual software engineering reasons, we find it convenient to divide the distributed grammar into modules; the grammatical knowledge associated with a device may reside in more than one module.

The grammar in our current demonstrator contains 21 modules, including the "core" grammar described above. Each device typically requires between two and five modules. For example, an on/off light switch loads three modules: the core grammar, the general grammar for on/off switchable devices, and the grammar specifically for on/off switchable lights. The core grammar, as already explained, consists of linguistically oriented device-independent grammar rules. The module for on/off switchable devices contains grammar rules specific to on/off switchable behaviour, which in general make use of the framework established by the general grammar. For example, there are rules of the schematic form

```
QUESTION -->
   is
   NP:[sem_type=device]
   ON_OFF_PHRASE
```

```
PARTICLE_VERB:[particle_type=onoff]
   --> switch
```

Finally, the module for on/off switchable lights is very small, and just consists of a handful of lexical entries for nouns like "light", defining these as nouns referring to on/off switchable devices. The way in which nouns of this kind can combine is however defined entirely by the on/off switchable device grammar and core grammar.

The pattern here turns out to be the usual one: the grammar appropriate to a device is composed of a chain of modules, each one depending on the previous link in the chain and in some way specialising it. Structurally, this is similar to the organisation of a piece of normal object-oriented software, and we have been interested to discover that many of the standard concepts of object-oriented programming carry over naturally to distributed unification grammars. In the remainder of the section, we will expand on this analogy.

If we think in terms of Java or a similar mainstream OO language, a major grammatical constituent like S, NP or PP has many of the properties of an OO interface. Grammar rules in one module can make reference to these constituents, letting rules in other modules implement their definition. For example, the temperature sensor grammar module contains a small number of highly specialised rules, e.g.

```
QUESTION -->
   what is the temperature
   PP:[pp_type=location]
QUESTION -->
   how many degrees is it
   PP:[pp_type=location]
```

The point to note here is that the temperature sensor grammar module does not define the locative PP construction; this is handled elsewhere, currently in the core grammar module. The upshot is that the temperature sensor module is able to define its constructions without worrying about the exact nature of the locative PP construction. As a result, we were for instance able to upgrade the PP rules to include conjoined PPs (thus allowing e.g. "what is the temperature in the kitchen and the living room") without in any way altering the grammar rules in the temparature sensor module[2]

In order for the scheme to work, the "interfaces" – the major categories – naturally need to be well-defined. In practice, this implies restrictions on the way we handle three things: the set of syntactic features associated with a category, the range of possible values (the domain) associated with each feature, and the semantics of the category. We consider each of these in turn.

Most obviously, we need to standardise the feature-set for the category. At present, we define most major categories in the core grammar module, to the extent of specifying there the full range of features associated with each category. It turns out, however, that it is sometimes desirable not to fix the *domain* of a feature in the core grammar, but rather to allow this domain to be extended as new modules are added. The issues that arise here are interesting, and we will discuss them in some detail.

The problems occur primarily in connection with features mediating sortal constraints. As we have already seen in examples above, most constituents will have at least one sortal feature, encoding the sortal type of the constituent; there may also be further features encoding the sortal types of possible complements and adjuncts. For example, the V category has a feature `vtype` encoding the sortal type of the V itself, a feature `obj_sem_np_type` encoding the sortal type of a possible direct object, and a feature `vp_modifiers_type` encoding the sortal type of a possible postverbal modifier.

Features like these pose two interrelated problems. First, the plug and play scenario implies that we cannot know ahead of time the whole domain of a sortal feature. It is always possible that we will connect a device whose associated grammar module requires definition of a new sortal type, in order to enforce appropriate constraints in the language model. The second problem is that it is still often necessary to define grammar rules referring to sortal features before the domains of these features are known: in particular, the core module will contain many such rules. Even before knowing the identity of any specific devices, general grammar rules may well want to distinguish between "device" NPs and "location" NPs. For example, the general "where-question" rule has the form

```
QUESTION --> where is NP
```

Here, we prefer to constrain the NP so as to make it refer only to devices, since the system currently

[2]An ambitious treatment of conjunction might arguably also necessitate changes in the dialogue management component specific to the temperature sensor device. In the implemented system, conjunction is uniformly treated as distributive, so "what is the temperature in the kitchen and the living room" is automatically interpreted as equivalent to "what is the temperature in the kitchen and what is the temperature in the living room'.

has no way to interpret a where question referring to a room, e.g. "where is the bathroom".

We have addressed these issues in a natural way by adapting the OO-oriented idea of inheritance: specifically, we define a hierarchy of possible feature values, allowing one feature value to inherit from another. In the context of the "where is NP" rule above, we define the rule in the core module; in this module, the sortal NP feature `sem_np_type` may only take the two values `device` and `location`, which we specify with the declaration[3]

```
domain(sem_np_type, [location, device])
```

This allows us to write the constrained "where is" rule as

```
QUESTION -->
   where is NP:[sem_np_type=device]
```

Suppose now that we add modules for both on/off switchable and dimmable devices; we would like to make these into distinct sortal types, called `switchable_device` and `dimmable_device`. We do this by including the following declarations in the "switchable" module:

```
domain(sem_np_type,
   [location,
    device,
    switchable_device])
specialises(switchable_device, device)
```

and correspondingly in the "dimmable" module:

```
domain(sem_np_type,
   [location,
    device,
    dimmable_device])
specialises(dimmable_device, device)
```

When all these declarations are combined at compile-time, the effect is as follows. The domain of the `sem_np_type` feature is now the union of the domains specified by each component, and is thus the set {`location`, `device`, `switchable_device`, `dimmable_device`}. Since `switchable_device` and `dimmable_device` are the precise values specialising `device`, the compiler systematically replaces the original feature value `device` with the disjunction

```
switchable_device \/ dimmable_device
```

Thus the "where is" rule now becomes

```
QUESTION -->
   where is
   NP:[sem_np_type=switchable_device \/
                   dimmable_device]
```

[3]We have slightly simplified the form of the declaration for expository purposes.

If new modules are added which further specialise `switchable_device`, then the rule will again be adjusted by the compiler so as to include appropriate new elements in the disjunction. The important point to notice here is that no change is made to the original rule definition; in line with normal OO thinking, the feature domain information is distributed across several independent modules, and the changes occur invisibly at compile-time[4].

We have so far said nothing about how we deal with semantics, and we conclude the section by sketching our treatment. In fact, it is not clear to us that the demands of supporting Plug and Play greatly affect semantics. If they do, the most important practical consideration is probably that plug and play becomes easier to realise if the semantics are kept simple. We have at any rate adopted a minimal semantic representation scheme, and the lack of problems we have experienced with regard to semantics may partly be due to this.

The annotated CFG grammars produced by our compiler are in normal Nuance Grammar Specification Language (GSL) notation, which includes semantics; unification grammar rules encode semantics using the distinguished feature `sem`, which translates into the GSL `return` construction. So for example the unification grammar rules

```
DEVICE_NOUN:[sem=light] --> light
DEVICE_NOUN:[sem=heater] --> heater
```

translates into the GSL rule

```
DEVICE_NOUN
   [ light {return(light)}
     heater {return(heater)}]
```

Unification grammar rules may contain variables, translating down into GSL variables; so for example,

```
NP:[sem=[D, N]] -->
   DET:[sem=D]
   NOUN:[sem=N]
```

translates into the GSL rule

```
NP (DET:d NOUN:n) {return(($d $n))}
```

Our basic semantic representation is a form of feature/value notation, extended to allow handling

[4]Readers familiar with OO methodology may be disturbed by the fact that the rule appears to have been attached to the daughter nodes (`switchable_device dimmable_device`, etc), rather than to the mother `device` node. We would argue that the rule is still conceptually attached to the `device` node, but that the necessity of eventually realising it in CFG form implies that it must be compiled in this way, so that it can later be expanded into a separate CFG rule for each daughter.

of conjunction. We allow four types of semantic construction:

- Simple values, e.g. `light`, `heater`. Typically associated with lexical entries.
- Feature/value pairs expressed in list notation, e.g. `[device, light]`, `[location, kitchen]`. These are associated with nouns, adjectives and similar constituents.
- Lists of feature/value pairs, e.g. `[[device, light], [location, kitchen]]`. These are associated with major constituents such as NP, PP, VP and S.
- Conjunctions of lists of feature/value pairs, e.g. `[and, [[device, light]], [[device, heater]]]` These represent conjoined constituents, e.g. conjoined NPs, PPs and Ss.

This scheme makes it straightforward to write the semantic parts of grammar rules. Most often, the rule just concatenates the semantic contributions of its daughters: thus for example the semantic features of the nominal PP rule are simply

```
NP:[sem=concat(Np, Pp)] -->
   NP:[sem=Np]
   PP:[sem=Pp]
```

The semantic output of a conjunction rule is typically the conjunction of its daughters excluding the conjunction itself, e.g.

```
NP:[sem=[and, Np1, Np2]] -->
   NP:[sem=Np1]
   and
   NP:[sem=Np2]
```

5 Future Plug & Play work

In the future, we intend to move to a system in which all dialogue components can be reconfigured by devices. For example, in a complete Plug and Play scenario, the possible device actions themselves should be declared by devices perhaps following UPNP standards in which devices publish all interface commands in the form `actionname(arg`$_1$ `...arg`$_i$`)` plus an internal state model of a simple vector of values. In this section we start with some very general observations on Plug and Play dialogue management and the role of inference. Then we outline a proposal for a rule based formalism.

At a very general level of course, indirections between executable actions and linguistic contents can arise at several levels: the speech act level ("It's too warm in here"), the content level ("How warm is it?"), as well through underdetermination of contents either through pronominal or elliptical constructions. At the moment, our pronominal and elliptical resolution methods depend on very simple 'matching' algorithms. In general, one might at least want some sort of consistency check between the linguistic properties expressed in an utterance and those of candidate objects referred to. One might expect that inferential elements in contextual interpretation should be strongly Plug and Play - they will depend, for correctness and efficiency, on tailoring to the current objects in the domain. The research project of uploading relevant axioms and meaning postulates from a device to a general purpose inference engine that can be invoked in contextual resolution looks very exciting.

Evidently, higher pragmatic relations between what the user "strictly says" and possible device operations are also very heavily inference based. At the moment, we simply encode any necessary inferences directly into the device grammars and this suffices to deal with certain simple behaviours. However, the requirement to encapsulate all device behaviour in a Plug and Play manner imposes a significant requirement. For example, the most natural interaction with a thermometer is, for example, "How warm is it?" or "What is the temperature?" and not "Query the thermometer". In our demonstrator, the (grammar derived) semantic values simply reflect directly the relevant device operations: ⟨ op=query device=thermometer⟩. The strategy supports the simple natural interactions it is designed to. It even interacts tolerably well with our ellipsis and reference resolution methods. "What is the temperature in the hall? And in the living room?" and "What is the temperature in the hall? What is it in the living room?" can both be correctly interpreted. Other interactions are less natural. The default output when resolution cannot identify a device N is "I don't know which N you mean". However, asking for the temperature in a room with several thermometers should probably not result in "I don't know which temperature you mean". It follows that prescribing all behaviour in a Plug and Play fashion is a significant constraint.

Indeed, a more general point can be made here. A problem has arisen because the inference from service required to service provider has become insecure in the presence of other service providers. In the highly networked homes of the future, more sophisticated inference may be required just because service level concepts will predominate over device level concepts.

5.1 A Rule based formalism

In this section we assume that semantic values consist of 5-ary slot-structure with slots `device_class` (dimmable light, TV ...), `device_attributes` (kitchen, blue ...), `pronoun` (as before) and `device-specifier` (as before) and `operation`. An operation is the action the user wants carried out, e.g. `switch_on`, `set_level(X)`, where `X` is a real number (for dimmable lights), `set_program(Y)` etc. (for Hi-Fis, TVs), and so on. As in the grammar, device classes are ordered in a hierarchy in a standard object-oriented way. Thus "dimmable light" is a subclass of "dimmable device" and inherits from it.

For strong plug and play, at least the following information must be loaded by a device into the dialogue manager: the device interface (e.g. that the "switchable light" class has a `switch_on` method; the feedback to the user; the update to the system's device model generated by executing the command. Clearly, behaviour can also depend on the current state. Reaction to "switch on the kitchen light" depends on whether the lamp is off, on, and whether there is a kitchen light. We write a rule as `command(A,B,C,D)` where `A` is a command, `B` is a class of devices for which A is applicable, `C` is a list of device attributes whose values must be known in order to execute A, `D` is a list of items describing how the system should react on A. Each item has the following components:

precondition(X) — `X` is an executable procedure that tests the network state and returns `true` or `false`. If `true`, the item can 'fire'.

action(Y) — `Y` is the procedure to execute

feedback(Z,R) — `Z` is the feedback for the user and can depend on `R`, the return value from the device operation

upd(W,R) — `W` describes how the system's model of the network state should be updated. Also `W` can depend on `R`.

For example switching on a light might be encoded as `command(switch, switchable_light, [id = ID], D)` where `D` is a list of items in the above form of which one describes behaviour when the light is already off thus:

```
[  precondition(light_off(ID)),
   action(switch_on_light(ID)),
   feedback(light_now_on(ID), success),
   feedback(could_not_switch(ID), error),
   upd([dev_update(ID, status = 1)], success),
   upd([], failure)  ]
```

`light_off` and `switch_on_light` are procedures provided by the lamp. The feedback to the user and the update rules depend on the result of the `switch_on_light` procedure.

6 Conclusion

Applying the idea of plug and play to spoken dialogue interfaces poses a number of interesting and important problems. Since the linguistic and dialogue management information is distributed throughout the network, a plug and play system must update its speech interface whenever a new device is connected. In this paper, we have focussed in particular on distributed grammars for plug and play speech recognition which we have integrated into our demonstrator system. We have also examined some issues and described a possible approach to distributed dialogue management which we plan to undertake in further work.

Acknowledgments

We are very grateful to our partners in the DHomme project for discussion of the above ideas - especially on the importance and role of differing strengths of Plug and Play. The DHomme project partners include netdecisions Ltd, SRI International, Telia Research AB, and the Universities of Edinburgh, Gothenburg and Seville.

References

N.M. Fraser and J.H.S. Thornton. 1995. Vocalist: A robust, portable spoken language dialogue system for telephone applications. In *Proc. of Eurospeech '95*, pages 1947–1950, Madrid.

Gerald Gazdar, Ewan Klein, Geoffrey Pullum, and Ivan Sag. 1985. *Generalized Phrase Structure Grammar.* Harvard University Press, Cambridge, MA.

J.R Glass. 1999. Challenges for spoken dialogue systems. In *Proc. IEEE ASRU Workshop, Keystone, CO.*

A. Goldschen and D Loehr. 1999. The role of the darpa communicator architecture as a human computer interface for distributed simulations. In *1999 SISO Spring Simulation Interoperability Workshop, Orlando, Florida, March 1999.*

SpeechWorks Int Inc, 2001. *SpeechWorks.* http://www.speechworks.com. As at 31/01/01.

B. Kiefer and H. Krieger. 2000. A context-free approximation of head-driven phrase structure grammar. In *Proceedings of 6th Int. Workshop on Parsing Technologies*, pages 135–146.

A. Kolzer. 1999. Universal dialogue specification for conversational systems. In *Proceedings of IJCAI'99 Workshop on Knowledge & Reasoning In Practical Dialogue Systems, Stockholm.*

S. Larsson and D. Traum. 2000. Information state and dialogue management in the trindi dialogue move engine toolkit. *Nat.Lang. Engineering*, 6.

Microsoft, 2000. *Universal Plug and Play Device Architecture.* http://www.upnp.org. Version 1.0, 8 June 2000.

D. Milward. 2000. Distributing representation for robust interpretation of dialogue utterances. In *Proc. of 38th ACL, Hong Kong*, pages 133–141.

R. Moore. 1998. Using natural language knowledge sources in speech recognition. In *Proceedings of the NATO Advanced Studies Institute.*

Nuance Communications, 1999. *Nuance Speech Recognition System Developer's Manual version 6.2.* 1380 Willow Road, Menlo Park, CA 94025.

S. Oaks and H. Wong. 2000. *Jini in a Nutshell.* O'Reilly.

Geoffrey K. Pullum and Gerald Gazdar. 1982. natural languages and context-free languages. *Llinguistics and Philosophy*, 4:471–504.

M. Rayner, J. Dowding, and B.A. Hockey. 2001a. A baseline method for compiling typed unification grammars into context free language models. Proc. Eurospeech 2001, Aalborg.

M. Rayner, G. Gorrell, B.A. Hockey, J. Dowding, and J. Boye. 2001b. Do cfg based language models need agreement constraints? In *Proceedings of 2nd NAACL*, Pittsburgh.

S. Sutton et al. 1998. Universal speech tools: The cslu toolkit. In *Proc. ICSLP-98*, pages 3221–3224.

Figure 1 - Example Dialogue

Network status: There is no TV. There is a VCR in the TV room.

U1	Switch on the television
S1	Sorry, try again
U2	Add new component (*this must precede device installation*)

Now, a TV is plugged into the living room. Currently we simulate this in a simple text interface, where devices and rooms are chosen from predefined lists. The speech interface is then automatically reconfigured (this currently takes 29 seconds)

U3	Switch on the television
S3	The TV is now on
U4	Switch on the television
S4	The TV is already on
U5a	Add new component

Now, another TV is plugged into the kitchen. No speech interface reconfiguration required this time

U5	Switch on the television
S5	The living-room TV is already on
U6	Switch on the VCR
S6	The VCR is now on
U7	Switch on the television
S7	Sorry, I don't know which TV you mean
U8	The television in the kitchen
S8	The kitchen TV is now on
U9a	Add new component

Now, on-off lights are installed in the hall and bathroom. Reconfiguration time is 27 seconds

U9	Switch on the light in the hall
S9	The hall light is now on
U10	Dim the light to fifty percent
S10	Sorry, try again.
U11a	Add new component

Now, a dimmable light is installed in the kitchen. Reconfiguration time is 39 seconds

U11	Dim the kitchen light to 50 percent
S11	The light in the kitchen is now on at fifty percent
U12	Dim the light in the hall
S12	I'm sorry, the hall light can only be on or off
U13	Turn on the light in the kitchen (misheard as "increase the light in the kitchen")
S13	The kitchen light is now on at sixty percent