Chapter 1

PLUG AND PLAY SPOKEN DIALOGUE PROCESSING

Manny Rayner[1], Johan Boye[2], Ian Lewin[1], and Genevieve Gorrell[1]

[1] *netdecisions Ltd*
Westbrook Centre,
Milton Road, Cambridge CB4 1YG, UK
manny.rayner|ian.lewin|genevieve.gorrell@netdecisions.com

[2] *Telia Research*
S-123 86 Farsta, Sweden
johan.x.boye@telia.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.

Keywords: Speech recognition, spoken language understanding, dialogue, device control, plug and play

1. Introduction

The notion of Plug and Play finds its most natural area of application 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. The discussion centres around a concrete prototype system, CANTONA, developed by netdecisions and Telia Research, which demonstrates plug and play spoken language control of a set of networked devices. To immediately give an intuitive idea of what we mean by plug and play functionality in this context, we start by presenting an annotated session with the current English language version of CANTONA (there is also a Swedish version). Spoken user utterances are prefixed with a "U", and spoken system responses with an "S". At the beginning of the dialogue, there is only one device connected, a VCR in the TV room:

U1: Switch on the television
S1: Sorry, try again

Since no television is currently connected, the system is not even able to recognise the word "television", and it is consequently unable to formulate any more helpful response than S1. We now connect a TV in the living room; currently we simulate the action of connecting a device using a simple text interface. CANTONA then automatically reconfigures iteself to include the new speech and language resources relevant to the TV component. The user's repeated request can now be understood, and results in the TV being switched on, together with an informative response:

U2: Switch on the television
S2: The TV is now on

We now connect a second TV, in the kitchen. When we repeat our initial request for the third time (U4), the system now engages in disambiguation behaviour (S4) whereas previously it had no need to:

U3: Switch on the VCR
S3: The VCR is now on
U4: Switch on the television

S4: Sorry, I don't know which TV you mean
U5: The television in the kitchen
S5: The kitchen TV is now on

In the next step, we connect lights with on/off switches in the hall and bathroom. U7/S7 show however that although the system can now respond to on/off commands, it is still unable to understand the word "dim":

U6: Switch on the light in the hall
S6: The hall light is now on
U7: Dim the light to fifty percent
S7: Sorry, try again

Finally, we connect a dimmable light in the kitchen. The system is now capable both of responding to correct dimmer commands (S8) and of providing helpful error messages in response to incorrect commands (S9):

U8: Dim the kitchen light to fifty percent
S8: The light in the kitchen is now on at fifty percent
U9: Dim the light in the hall
S9: I'm sorry, the hall light can only be on or off

Simple as it is, the above dialogue illustrates the main points of plug and play spoken language dialogue. At any given moment, the system's dialogue capabilities are determined by the set of devices currently connected; adding new devices dynamically changes its ability to recognise, understand, and respond to commands. It is still however impossible to determine from the dialogue alone how this behaviour is being produced; at this point, we will find it helpful to introduce a little terminology, and distinguish between *weak* and *strong* plug and play functionality.

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 still, of course, 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.

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".

The notion of Plug and Play has also 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.

We believe our notion of Plug and Play is an appealing theoretical one because it embodies an interestingly different notion of knowledge modularity and a new perspective on the idea of system reconfigurability. Of course, re-configuration of spoken language systems has long

been a goal of language engineering. However, it 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.

In the rest of the paper, we will primarily be concerned with strong plug and play functionality as it is manifested in a rule-based spoken dialogue system like CANTONA. Section 2 gives an overview of the CANTONA system, describing the top-level components and the key interfaces. The meat of the paper in in the next three sections. Section 3 describes general architectural considerations relevant to achieving plug and play functionality in rule-based systems; Section 4 describes how this kind of architecture can be realised for the case of dialogue management; and Section 5 describes how it can be realised for speech recognition and parsing. The last section suggests directions for further work and concludes.

2. The CANTONA Plug and Play Demonstrator

This section provides an overview of the CANTONA demonstrator. All processing is rule-based; the fundamental idea is to realise plug and play functionality by associating each device with its own set of rules, together with a core device-independent rule-set.

The system comprises four main components, respectively responsible for speech understanding, dialogue management, action management and spoken output generation, with low-level communication handled over sockets. We briefly describe each of these components, focussing on top-level functionality and interfaces.

Speech understanding The speech understanding module is implemented on top of the Nuance Toolkit platform [Nuance Communications, 2002], using the C-based DialogBuilder API, and carries out the tasks of speech recognition, parsing, and construction of surface semantic representations. The linguistic knowledge used by the module is encoded as a unification grammar, and compiled first

into a representation in Nuance Grammar Specification Language (GSL) and then into a Nuance recognition package. The module's output consists of context-independent semantic structures in a slightly extended attribute/value notation described below.

The speech processing module has strong plug and play functionality, with each device contributing a piece of unification grammar relevant to its specific functionality.

Dialogue management The dialogue management component is implemented in SICStus Prolog [SICStus, 2000], and is responsible for deep semantic processing, including contextual interpretation of the user's command, translation into device instructions, and response generation. It accepts the semantic structures produced by the speech understanding module, and produces requests for the action management and spoken output generation modules.

The dialog management module also has strong plug and play functionality, with each device contributing a set of device-specific rules.

Action management The action management module manages the low-level interaction with real or simulated devices; it translates abstract device commands created by the DM into device-specific form, sends them to the devices, receives and decodes messages from the devices, reports errors that may occur during the execution of instructions, and so on. CANTONA can be configured to control both simulated and real devices; real devices are controlled through a LonWorks network [LonWorks, 2002] via a Java servlet.

The action management module also has strong plug and play functionality.

Spoken output generation The spoken output generation module receives abstract generation requests from the dialog manager, and transforms them into spoken responses using a simple *ad hoc* surface generation strategy; speech is produced by playing pre-recorded audio files through the Nuance Toolkit's standard interface.

The spoken output generation module currently has no plug and play functionality.

The central level of representation is the semantic form sent from the speech understanding module to the dialogue management module. User utterances are encoded as feature-value lists, extended to allow handling of conjunction. The semantic form representation is described in detail at the end of Section 5; for the moment, we content ourselves with some representative examples.

"Switch the kitchen light on"
```
[[operation, switch_on], [device, light], [spec, the], [location,
kitchen]]
```

"Switch on the light and the heater in the kitchen"
```
[[operation, switch_on], [and, [[device, light], [spec, the]],
[[device, heater], [spec, the]]], [location, kitchen]]
```

"Dim the living-room light to fifty percent"
```
[[operation, dim_to], [onoff_level, 50], [device, light], [spec,
the], [location, living_room]]
```

"Switch it off"
```
[[operation, switch_off], [pron, it]]
```

"And the one in the kitchen"
```
[[device, undefined], [location, kitchen]]
```

The dialogue manager's design is essentially an elaboration of the output/meta-output architecture described in [Rayner et al., 2000]. It receives surface semantic representations from the speech understanding module, and processes them in turn through the two phases of *resolution* and *response generation*. The resolution phase attempts to translates the input semantic representation into a deep semantic representation, which we call *an executable form*; this may involve locating referents for descriptions and pronouns, or interpretation of ellipsis. The response generation phase accepts executable forms, and attempts to act on them. Possible ways to respond include carrying out commands (with or without accompanying verbal confirmation), answering questions, or giving various kinds of informative feedback if error conditions occur.

Looking at last three examples immediately above, for instance, the representation of "Dim the living-room light to fifty percent" will be translated into the form

```
[dim(50), <Dev1>]
```

if a device with ID `<Dev1>` can be located which has the properties

```
[[device, light], [location, living_room]]
```

The resource used to locate the device is a database called the *world-state*, which contains information about all the devices currently connected to the system. If no such device can be found in the world-state database, or if there are several such devices, resolution instead passes an error message to response generation describing the nature of the problem.

Assuming that a suitable device was found in the world state, and that processing was thus successful, the representation of the follow-up utterance "Switch it off" will be translated into the form

```
[switch_off, <Dev1>]
```

since the component

```
[pronoun, it]
```

can be resolved to the object referred to in the previous utterance. The final utterance, "And the one in the kitchen" will be translated into the form

```
[switch_off, <Dev2>]
```

if a device with ID `<Dev2>` can be located which has the properties

```
[[device, light], [location, kitchen]]
```

In this case, the resolution component fills in the missing semantic values for `operation` and `device` from a set of defaults maintained by the current discourse context. Again, if no such device can be found, the result is that an appropriate error message is passed to response generation instead.

The response generation module accepts executable forms from the resolution module, and attempts to act on them. Just as in the case of resolution, execution may be successful or unsuccessful. For example, successful processing of our first executable form,

```
[dim(50), <Dev1>]
```

will result in the light `<Dev1>` being dimmed to an intensity of 50%, together with production of a confirmation utterance. Processing may on the other hand be unsuccessful for several possible reasons. The device in question might be inoperational, or inaccessible due to network problems, or disconnected, or it could be an on/off light rather than a dimmer. Thus some problems may arise immediately, while others can arise only when the result of the operation is reported back from the physical device. In each case, the response generation module has rules which associate different failure types with appropriate feedback messages, resulting in spoken output to the user describing the reason why the operation failed. The form of these rules is described in detail in Section 4.

3. Device encapsulated rules and grammars

In order to realise Plug and Play in a rule-based spoken dialog system, we must associate each device with a set of rules. These rules will be uploaded when the device is connected, and compiled together with rules from other devices and also, in general, a set of device-independent rules. CANTONA currently admits two types of device-dependent rules. Linguistic rules define the constructions (words and phrase-types) specific to the device. They are used by the speech understanding component and described in Section 5. Response generation rules define the behaviour of the response generation subcomponent of the dialogue

manager, and are described at the end of Section 2. At present, the rules used by the resolution subsystem of the dialogue manager are all device-independent.

Rules on their own are of course valueless; they must be compiled into executable code or interpreted. These aspects of the system are however of less interest to us here, since they have no direct relevance to the central theme of achieving Plug and Play functionality, and in the sequel we will more or less take them for granted. We will instead focus on what we see as the central question: how to structure our rule-sets so that they can be distributed between the various components.

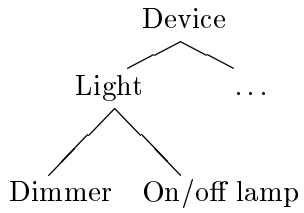
Consider a concrete example. Suppose that a light with a dimmer switch is added to the network. The speech understanding component is updated with the necessary words and phrases, including nouns like "light" and "lamp", which directly refer to that type of device. There will also be grammar rules permitting use of constructions like "Dim X", "Set X to Y percent", "Switch X off", and so on. Many grammar rules will have a much wider area of applicability than just dimmer lights. The word "light" pertains only to one or two specific types of device; but "Switch X off" is applicable much more broadly. It is highly desirable that the knowledge used by the system should not only be distributed over the various devices, but also be *hierarchically organised*. In fact, we can imagine at least three different hierarchies over

1 the linguistic resources needed to query & control devices

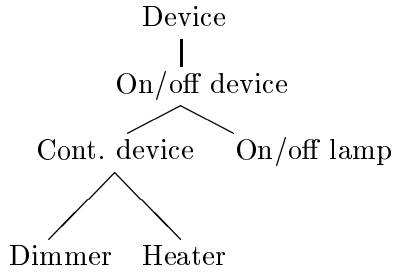
2 the functionalities the devices implement, or

3 the code needed to control the devices.

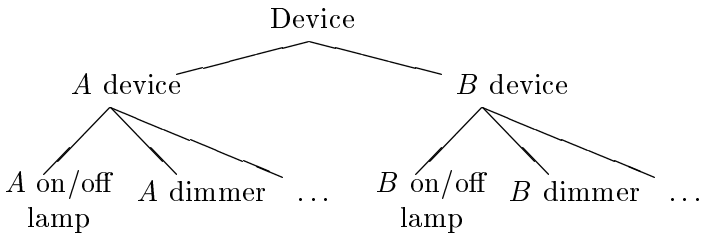
In the first case above, it seems reasonable that lights that can only be turned on or off ("switchable" lights) should be instances of a more general "light" class. Sentences like "Is the lamp in the kitchen on?" or "Switch on the light in the hall" could refer to dimmers or switchable lights. (This is even more evident in Swedish where there is a special verb *tända* that can refer to lamps but not other switchable devices like TVs or CD players). Thus according to this principle, a part of the device hierarchy would look as follows:



Considering functionality however, it seems that "continously controllable" devices, of which dimmers form a subclass (together possibly with heaters, fans, and so on), constitute a subclass of on/off-devices. Dimmers implement all the operations of an on/off device, in addition to the operation "dim". (Switching on a dimmer could be understood as either turning the illumination level to 100%, or to the level the dimmer was set to before it last was switched off). Thus we have the following situation:



Thirdly, consider the situation where there are several manufacturers supplying the hardware and software needed to actually control the physical devices (by sending control pulses over the electricity network, for instance). We might reasonably assume that the software controlling many of manufacturer A's devices would overlap to some extent, i.e. that A would have general device-control software inherited by several devices. Thus, a third possibility is to organise the devices according to the code base needed to access and control them.



So which is the correct view? In our opinion, they all are; in order to facilitate reusability and maintainability of the plug-and-play spoken-dialogue interface, it is important to let objects (= device descriptions) inherit linguistic resources, dialogue descriptions and code. In principle, it is possible to collapse all hierarchies into one, by taking the cross-product of the above diagrams. But that would only confuse the issue, not only because resulting diagram would be complicated, but because the diagrams above represent three different inheritance relations.

The challenge is to define a computational formalism that supports these three inheritance relations, avoids the problems associated with multiple inheritance, and is straightforward to use. This is the subject of the next two sections. The basic idea is to associate each device with four distinct pieces of information: an *identifier*, a *class*, a set of *attributes*, and a *grammar*. A typical device declaration is written as follows:

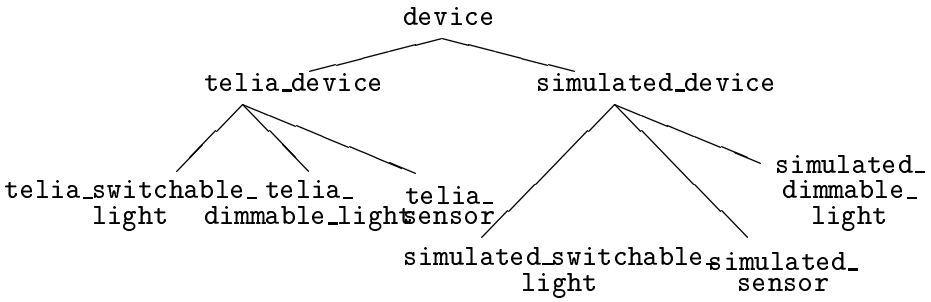
```
device(
  'LIGHT3',
  attributes([location=kitchen, level=0.0]),
  class(telia_switchable_light),
  grammar(switchable_light)
).
```

Here `LIGHT3` is the identifier of the device, possessing the attributes `location` and `level` with initial values `kitchen` and `0.0` respectively; these attributes are used to update the dialog manager's world-state sub-component.

The types of executable form accepted by the device, and the way in which they are handled by response generation, are determined by its class, `telia_switchable_light`. Classes are described in detail in Section 4. Finally, the device extends the speech understanding component with the words and constructions listed in the grammar module `switchable_light`. Grammar modules will be described in Section 5.

4. Plug and Play Response Generation

This section describes the Plug and Play response generation component, whose fundamental construct is the *class definition*. A class definition contains the code for accessing devices belonging to that class. Classes are arranged in an inheritance hierarchy; so for example `telia_switchable_light`, `telia_dimmable_light` and `telia_sensor` are all subclasses of `telia_device`. The latter class contains code for accessing a device in the Telia intelligent-home demonstration environment (the "Vision Centre"), while its different subclasses contain more specialised code for performing specific operations (like "dim") on a device. The diagram below shows the class hierarchy of our current demonstrator.



From the point of view of dialogue management, however, instructing a device to perform an operation is the same, regardless of the environment of the device, or of whether the device is real or simulated. The same information has to be given to the system, the same kind of ambiguities may arise, the system will ask the same kind of clarification questions, and so on. Thus, there is a relationship between classes like `telia_switchable_light` and `simulated_switchable_light` which is not shown by the class diagram above. And, as already discussed, adding more superclasses and arrows would lead to a diagram with multiple inheritance and an unclear operational interpretation. To avoid these problems, we will borrow an idea from the programming language Java.

What the two classes `telia_switchable_light` and `simulated_switchable_light` have in common is that their instances may perform the operations `switch_on` and `switch_off`. We will say, analogous to Java, that the two classes *implement the interface* `switchable`.

Thus the class definition for `telia_switchable_light` begins as follows:

```
class(
  telia_switchable_light,
```

```
  extends( telia_device ),
  implements( switchable )
).
```

indicating the position of `telia_switchable_light` in the class hierarchy, but also listing the set of interfaces the class is implementing.

Parts of the interface declaration for `switchable` is shown in figure 1.1. (The declaration also contains a specification of the operation `switch_off` which is not shown.) An interface declaration contains one or more operation declarations. An operation declaration is a quadruple

$$\langle \texttt{name}, \texttt{interface}, \texttt{id}, L \rangle$$

where `id` is a variable acting as a placeholder for the identifier of the device which is to perform the operation. L is the meat of the definition; it is a list of rules specifying the system's dialogue behaviour associated with the operation.

A rule has four kinds of elements:

condition(P) — `P` is a pre-condition of the rule; if `P` is true, then this rule will be fired upon consideration. If `P` is false, the rule will not be fired.

action(A,R) — `A` is the procedure call that should be executed if the precondition above is true. The optional parameter `R` holds the result sent back from the device after execution of `A`.

feedback(C,F) — `F` is the feedback that should be given to the user when the action `A` has finished executing and the result `R` is returned, provided that the condition `C` is true. In most cases, `C` will be dependent on `R`. `C` is an optional parameter; if omitted, it is assumed to have the value `true`.

update(C,U) — `U` describes how the system's model of the network state should be updated when the action `A` has finished executing and the result `R` is returned. The update instruction `U` will only be executed if the condition `C` is true. In most cases, `C` will be dependent on `R`. `C` is an optional parameter; if omitted, it is assumed to have the value `true`.

The operational interpretation of a rule is as follows. First, the precondition is evaluated. If the result is `false`, nothing more happens. If the result is `true`, the action `A` is executed, and upon its termination, the feedback items and update items are evaluated, as follows:

If there are several feedback items, their preconditions are evaluated in top-to-bottom order, and the first feedback with a true precondition

```
interface( switchable ).

operation(
  switch_on,
  switchable,
  DevID,
  [
    [
      condition( and(device_connected(DevID),
                     device_switchable(DevID),
                     device_off(DevID)) ),
      action( switch_on(DevID), R ),
      feedback( action_successful(R),
                device_has_been_switched_on(DevID) ),
      feedback( action_unsuccessful(R),
                device_has_not_been_switched_on(DevID) ),
      update( action_successful(R),
              device_update(DevID, level=1.0)) ,
      update( action_unsuccessful(R), no_update )
    ],

    [
      condition( and(device_connected(DevID),
                     device_switchable(DevID),
                     device_on(DevID)) ),
      action( no_action ),
      feedback( device_is_already_on(DevID) ),
      update( no_update )
    ],

    [
      condition( device_not_connected(DevID) ),
      action( no_action ),
      feedback( device_not_connected(DevID) ),
      update( no_update )
    ],

    [
      condition( device_not_switchable(DevID) ),
      action( no_action ),
      feedback( device_not_switchable(DevID) ),
      update( no_update )
    ]
  ]
).
```

Figure 1.1. Parts of the `switchable` interface definition

(or with no precondition) is produced as the system's output to the user. The remaining feedback items are *not* evaluated. Update items are treated analogously.

As an example, consider the first rule in the declaration for `switch_on` in Fig. 1.1. This rule is fired if the device with the identity `DevID` is connected to the network, if it is indeed a switchable device, and if it is currently switched off (we will return to *how* these conditions are evaluated in the next section). If the rule is fired, the system instructs the device with id `DevID` to switch on (this is specified by the `action` item). After the device has carried out the `switch_on` operation, the system generates feedback and updates its internal state, depending on the return value `R` sent from the device.

Note that, within a rule, there may be any number of feedback and update items, but there may only be one precondition and one action item.

Any item may be omitted within the rule. If the precondition is omitted, it is regarded as true, and the rule will be fired upon consideration. An alternative way is to write `condition(true)`. If there is no action item, no action will be performed as a result of executing this rule. An alternative way of specifying this is `action(no_action)`, as in the example in Fig. 1.1.

The operational interpretation of an operation declaration, such as the one in Fig. 1.1, is as follows. The preconditions of the rules are evaluated, in top-to-bottom order, and the first rule with a true precondition (or with no precondition) is executed.

4.1 The interface-class relationship

We now turn to the relationship between interfaces and classes. As noted from the example, an interface expresses the operational behaviour of the system in terms of conditions, actions and update operations. The interface does not describe, however, how conditions are evaluated, and how actions and update operations are performed. This is the purpose of the classes.

For a class to implement an interface, it must provide code for evaluating all the conditions, and executing all actions and update rules, that are referred to in the interface. Thus, the two classes `telia_switchable_light` and `simulated_switchable_light` (or their superclasses) both provide code for evaluating `device_connected(DevID)`, `device_on(DevID)`, and so on, as well as code for executing `switch_on(DevID)` and `device_update(DevID, level=1.0)`, and so on. They

do so, however, *in two completely different ways*, since one class deals with real switchable lights, and the other with simulated ones.

This division between interfaces and classes thus has several advantages:

- It allows a clear division between high-level dialogue and contextual interpretation rules on the one hand (interfaces), and low-level implementation details on condition evaluation and command execution on the other (classes).

- It avoids problems related to multiple inheritance by distinguishing between a class hierarchy and an interface hierarchy (as outlined in Section 3).

- It allows the user to overload natural commands like "switch off". Switching off a lamp and switching off a computer are two very different things (and thus the lamp and the computer will implement them very differently), but to the user they seem the same: She can just say "Switch off the lamp" or "switch off the computer", and it will work as intended.

5. Plug and Play Speech Recognition and Parsing

In this section, we 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 provides 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.

Modern speech interfaces supporting complex commands are typically specified using a rule-based grammar formalism defined by a platform like Nuance [Nuance Communications, 2002] or SpeechWorks [SpeechWorks, 2002]. 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 [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 directly relevant to the "distributed" unification grammars needed for Plug and Play.

We start with a general device-independent unification grammar, which implements the core grammar rules. In our current English language 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 the methods in 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[1]

In order for the scheme to work, the "interface methods" – 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 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[2]

```
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]
```

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[3].

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 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]
```

Some examples of semantic representations can be found in Section 2.

6. Discussion

The focus of our paper has been on adding Plug and Play functionality to a complex rule-based system, but there are of course other possible types of architecture where the Plug and Play idea is potentially applicable. For example, in a very simple system the command vocabulary offered by the speech interface may just consist of a list of fixed phrases, and dialog management may make no reference to context. 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 command grammar; a similar approach can be used for the dialog management component.

Another popular type of architecture performs recognition using a robust statistical recogniser, and language-processing through device-specific phrase-spotting rules, e.g. [Milward, 2000]. It seems possible in principle to add Plug and Play functionality to a system of this kind too, but the problems which arise are rather different in nature; for example, it is not immediately clear how to modularise a statistical language model so as to give it Plug and Play functionality. Of course, a system with a large vocabulary recognizer can simply represent weak plug and play with respect to recognition and possibly strong plug and play in dialogue management. Evidently, exploring the space of possible combinations represents a fascinating research area in its own right. Indeed, the "right combination" may depend on many factors - including the current state of component technologies. One of our objectives has been to investigate the reconfiguration of a rule based speech recognizer, precisely because it currently represents the commercially most attractive recognition solution.

The role of "modular device knowledge" in inference based processes is another area we have not addressed at all. For example, 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.

Furthermore, 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. The most natural interaction with a thermometer is "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 and 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. On the other hand, the default output when resolution cannot identify a device N is "I don't know which N you mean". Asking for the temperature in a room with several thermometers should probably not result in "I don't know which temperature you mean". Clearly, 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 such service level concepts will predominate over device level concepts.

Acknowledgments

Plug and play spoken dialogue functionality formed a central theme of the D'Homme project (EU 5th Framework project IST-2000-26280),

and much of the work described in this paper was funded under it. We are grateful to our partners in the D'Homme project for many useful discussions, especially concerning the importance and role of differing strengths of Plug and Play.

Notes

1. 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'.

2. We have slightly simplified the form of the declaration for expository purposes.

3. 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.

References

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

[Gazdar et al., 1985] Gazdar, G., Klein, E., Pullum, G., and Sag, I. (1985). *Generalized Phrase Structure Grammar*. Harvard University Press, Cambridge, MA.

[Glass, 1999] Glass, J. (1999). Challenges for spoken dialogue systems. In *Proc. IEEE ASRU Workshop, Keystone, CO*.

[Goldschen and Loehr, 1999] Goldschen, A. and Loehr, D. (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*.

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

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

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

[LonWorks, 2002] LonWorks (2002). *LonWorks Device Control* http://www.lonworks.com. Version x.x, 15 February 2002.

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

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

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

[Nuance Communications, 2002] Nuance Communications (2002). *Nuance Speech Recognition System Developer's Manual*. 1380 Willow Road, Menlo Park, CA 94025.

[Oaks and Wong, 2000] Oaks, S. and Wong, H. (2000). *Jini in a Nutshell*. O'Reilly.

[Rayner et al., 2000] Rayner, M., Hockey, B., and James, F. (2000). A compact dialogue management architecture based on scripts and meta-outputs. In *Proc. 5th Conference on Applied Natural Language Processing, Seattle*.

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

[Rayner et al., 2001b] Rayner, M., Gorrell, G., Hockey, B., Dowding, J., and Boye, J. (2001b). Do CFG based language models need agreement constraints? In *Proceedings of 2nd NAACL*, Pittsburgh.

[SICStus, 2000] SICStus team. (2000). *SICStus Prolog User's Manual*. Swedish Institue of Computer Science.

[SpeechWorks, 2002] SpeechWorks (2002). SpeechWorks. http://www.speechworks.com. As at 15 February 2002.

[Sutton et al, 1998] Sutton et al, S. (1998). Universal speech tools: The CSLU toolkit. In *Proc. ICSLP-98*, pages 3221–3224.