D2.2: Dynamic Multimodal Interface Reconfiguration

David Milward (ed), Gabriel Amores, Nate Blaylock, Staffan Larsson, Peter Ljunglöf, Pilar Manchón, Guillermo Pérez

Distribution: Public

TALK
Talk and Look: Tools for Ambient Linguistic Knowledge
IST-507802 Deliverable D2.2

8th August 2006

Project funded by the European Community under the Sixth Framework Programme for Research and Technological Development

The deliverable identification sheet is to be found on the reverse of this page.

Project ref. no.	IST-507802
Project acronym	TALK
Project full title	Talk and Look: Tools for Ambient Linguistic Knowledge
Instrument	STREP
Thematic Priority	Information Society Technologies
Start date / duration	01 January 2004 / 36 Months

Security	Public
Contractual date of delivery	M30 = June 2006
Actual date of delivery	8th August 2006
Deliverable number	D2.2
Deliverable title	D2.2: Dynamic Multimodal Interface Reconfiguration
Type	Report
Status & version	Final 1.1
Number of pages	64 (excluding front matter)
Contributing WP	2
WP/Task responsible	LING
Other contributors	UGOT, USE, USAAR,
Author(s)	David Milward (ed), Gabriel Amores, Nate Blaylock, Staffan Larsson, Peter Ljunglöf, Pilar Manch´on, Guillermo P´erez
EC Project Officer	Evangelia Markidou
Keywords	ontology, reconfigurability

The partners in TALK are:	**Saarland University**	USAAR
	University of Edinburgh HCRC	UEDIN
	University of Gothenburg	UGOT
	University of Cambridge	UCAM
	University of Seville	USE
	Deutches Forschungszentrum fur Künstliche Intelligenz	DFKI
	Linguamatics	LING
	BMW Forschung und Technik GmbH	BMW
	Robert Bosch GmbH	BOSCH

For copies of reports, updates on project activities and other TALK-related information, contact:

The TALK Project Co-ordinator
Prof. Manfred Pinkal
Computerlinguistik
Fachrichtung 4.7 Allgemeine Linguistik
Postfach 15 11 50
66041 Saarbrücken, Germany
pinkal@coli.uni-sb.de
Phone +49 (681) 302-4343 - Fax +49 (681) 302-4351

Contents

Summary

Wider use of spoken and multi-modal dialogue systems has been hampered by the cost of reconfigurability to new tasks and applications. Dynamic reconfiguration offers the potential for immediate redeployment, and the possibility of more personalised and context dependent interaction with users. This deliverable outlines the challenges for knowledge representation, and how the systems address various levels of reconfigurability up to and including dynamic reconfigurability.

Chapter 1

Introduction

In this report we discuss the steps that have been taken towards fully reconfigurable multi-modal dialogue, from greater use of declarative domain representations, up to fully dynamic reconfiguration where a dialogue manager can be updated while on-line.

We focus on the home information and control scenario which provides a particularly interesting, but also particularly challenging scenario for dialogue systems. Each home is different, with different numbers of rooms, different numbers of floors, different kinds of devices, and different devices in each room. Although it is possible to have separate dialogue systems for each device, this kind of approach loses many of the benefits of spoken interaction. Ideally we want:

1. The ability to control and interrogate more than one device at a time e.g. *turn off all the lights*, *have I left anything on downstairs?*
2. A uniform interaction with each device (users do not want to learn a new style of interaction for each device).
3. Guided interaction which does not have to be learnt (this contrasts with having to teach the system a particular exact command for every possible action on every device).
4. Access to multiple devices so that these can be programmed together; e.g. *turn on the hall light when the door is opened*. Programming using dialogue will be discussed in detail in TALK Deliverable D2.3.
5. Automatic integration of new devices in the home set–up.
6. Easy ways to configure the system for different homes, and to reconfigure as devices are moved around the home.
7. Different levels of access and control for different users.

When we consider home information or services, the challenges are greater, but so are the potential benefits. Each user in the same household may subscribe to their own particular set of services. Ideally we want:

1. Information to be shared across services: we do not want to have to repeat a logon, billing information, etc, for every service.

2. A uniform approach to interaction with each service, which does not have to be learnt.
3. Setting of defaults on services e.g. *whenever I go to London try Stansted first*. This is again discussed in D2.3.
4. Proactivity: when the circumstances change or new events come into play, the user/s may want to change the regular routines.
5. Easy integration of new services.

Considering spoken dialogue for home control and home services as a whole, the key requirements are:

1. The home is aware of itself, its devices, the services available, and the full set of users and their access levels and preferences. That is, it can handle all relevant information.
2. The home can easily adapt to changes and additions (new services and/or devices).
3. The home interacts naturally at all levels.

Although a few words can be used to summarize the overall goal (self–awareness, adaptability and natural interaction), there is a great deal of complexity implied in this list. This complexity grows as we consider devices and services as interrelated entities that may need or should *talk* to each other and/or exchange information. Another factor that adds complexity is the existence of several users in the same house. This is challenging not just because of the existence of several user profi les, but because of the possibility of two different users interacting with the home simultaneously through the same or different modalities in different locations (eg.: two users in different rooms of the same house accessing the system, or one user located in the house and another user accessing the house control remotely though the telephone, internet, etc.). Although it is possible to use a new dialogue manager process for each user, with no explicit synchronisation except through external events, it might also be convenient for the dialogue manager to have some global knowledge of who is communicating with the system and what they are doing so that users can be warned, if, for example, another user is currently in a dialogue with the same device.

Multimodal interfaces add yet more complexity, as well as a greater degree of flexibility to users. The availability of more than one communication channel not only gives the possibility of splitting a single task across different modalities, but allows for the possibility of different tasks to be undertaken simultaneously in different modalities. Multimodality thus requires ability to deal with:

1. Input modality integration (fusion)
2. Choice of output modalities or combination (fi ssion)
3. Multitasking

To sum up, in order to provide a multi-modal solution that copes with all the tasks and requirements that we would ideally include for practical and appealing home control systems, several concepts must be taken into account:

1. Smart house self-awareness: full information about devices, services, house confi guration, device distribution, etc. in standardized formats to ensure compatibility.

2. Flexible and natural interaction in multimodal environments.
3. Adaptable and collaborative systems to ensure easy reconfiguration and integration of new devices and services.
4. Simultaneous conversations with different users.
5. Multimodal Multitasking.

The complexity outlined above makes it difficult to use traditional dialogue systems with largely pre-scripted interactions which would have to predict all the possible ways in which different devices interact and how this might affect the interaction with the user through different modalities. A new approach is therefore required. In this report we describe approaches which are rooted in the Information State Update approach, but also include a knowledge-based (or *o*ntology-based) approach to the incorporation of domain specific knowledge, separating this from general multi-modal dialogue behaviour.

Chapter 2 is concerned with how knowledge about devices and services is represented in the systems, for example using the OWL ontology representation language. Chapter 3 is concerned particularly with dialogue management aspects, including distribution of control between a dialogue manager and external tasks. Chapter 4 focusses on issues of dealing with multiple applications within a single system. Chapter 5 focusses on the practicalities of reconfiguring each of the system and categorises them according to the degree to which you can plug-and-play devices and services at will.

Chapter 2

Representing Application and Domain Specific Knowledge

2.1 Introduction

To fully abstract application and domain specific knowledge from generic dialogue behaviour, we need to be able to specify knowledge required by the dialogue manager concerning:

1. Individual Devices
2. Individual Services
3. The general context (the home in this scenario)
4. Any required grouping or menu structure to access devices or services
5. Any communication required between the dialogue manager and the external devices

The use of a similar representation for the first four cases is desirable since there is considerable overlap between the four knowledge areas. A device may be associated with device specific functions which may be regarded as services, so Case 1 and 2 overlap. It may also be useful to group specific functions of a device into menu structures, so Case 1 and Case 4 overlap. Furthermore, the context for a component within a device may also resemble the context for a single device within the home, giving overlap between Case 1 and Case 3.

In this chapter we therefore concentrate on the use of general description frameworks, in particular the use of ontologies and the CPS model. This does not mean that the systems are incompatible with standards which are specifically for devices e.g. UPNP (Universal Plug and Play), or for services e.g. OWL-S (OWL-based Web service language), or user interfaces e.g. UIDL (User Interface Description Language). In fact, it makes sense to map from each of these descriptions into a more general representation which can include information about device and service functionality, but also lexical information which says how to refer to the functions or components of the device. In the case of GoDiS there is an explicit description of the interface to UPNP. Further information on mapping from UIDL descriptions is given in TALK Deliverable D2.1 [MAB+05].

2.2 Representing a Home Ontology in OWL

One of the main objectives of the work in Seville was to identify a broadly used standard for the specification of ontological knowledge, that would:

- Enable partners to work in the same ontological framework
- Allow for external contributions and sharing of ontological knowledge.
- Bring the TALK project work closer to other international projects, and
- Take advantage of already developed work on this issue.

2.2.1 Ontology Coverage

An additional objective within this task was the design of a more exhaustive and complete version of the home ontology, which should incorporate the "telephone operator" functionality defined in Siridus [Sir] For this purpose, new devices and relations have been included and a new ontological structure has been designed. Obviously, the ontology described below is just illustrative, and does not mean to be exhaustive, neither in its class coverage nor in the individuals included in each class or subclass.

2.2.2 Classes and Subclasses

The basic element in OWL consists of the triplet Subject–Predicate–Object. Subjects and objects are denoted by classes and subclasses, while Predicates are typically denoted by properties.

System is a special class, whose functionality will be described below.

Device: Describes device types, and contains the following subclasses:

1. **Lamp** and **Dimmer** (actually a subclass of **Lamp**)
   (a) *Lamp_1* through *Lamp_6* as individuals
   (b) *DimmerLamp_1* and *DimmerLamp_2* individuals
2. **Blind** (one individual, *Blind_1*)
3. **Fan** (one individual, *Fan_1*)
4. **Heater** (one individual, *Heater_1*)
5. **Radio** (one individual, *Radio_1*)
6. **Sensor** (one individual, *Sensor_1*)
7. **TV** (one individual, *TV_1*)

Area includes the following individuals:

- *Upstairs*
- *Downstairs*

- *Indoors*
- *Outdoors*

Room includes eight subclasses:

1. **Bathroom** (*Bathroom_1* and *Bathroom_2*)
2. **Bedroom** (*Bedroom_1* through *Bedroom_4*)
3. **Garage** (*Garage_1*)
4. **Garden** (*Garden_1*)
5. **Hall** (*Hall_1*)
6. **Kitchen** (*Kitchen_1*)
7. **Patio** (*Patio_1*)
8. **Sittingroom** (*Sittingroom_1*)

SpecificLocation contains 5 sample individuals:

- *Ceiling*
- *LeftCorner*
- *Table*
- *Wall*
- *RightCorner*

Color is a class available for both rooms and devices, whose individuals are:

- *Black*
- *Blue*
- *Green*
- *Red*
- *White*
- *Yellow*

Size has two individuals:

- *Big*
- *Small*

The actual functionality taken into account in this new version of the ontology includes the following commands for the home devices:

- Switch on / off
- Open / Close

- Raise / Lower

As discussed previously, the telephone operator functionality and its corresponding directory of entries have also been integrated in the new ontology. However, they deserve a special consideration within the ontology with respect to the rest of the devices and/or functionalities. Ontologically, a class of name **Directory** describes the directories available in the home (currently, it only contains one individual).

The information stored in each entry in the directory is defined as a **DirectoryEntry** class, with six individual instantiations in our example (*DirectoryEntry_1* to *DirectoryEntry_6*). Each Directory Entry requires a set of objects, defined as independent classes:

- **FirstName** (several instances)
- **LastName** (several instances)
- **HomeNumber** (no instances)
- **WorkNumber** (no instances)
- **Mobile** (no instances)
- **Email** (20 individuals)
- **Relationship** to the user, with the following individuals:
  - *Boss*
  - *Brother*
  - *Cousin*
  - *Father*
  - *Friend*
  - *Mother*
  - *Neighbour*
  - *Sister*
  - *Uncle*

2.2.3 Properties

Instances of objects are linked to other objects according to the OWL triplet by means of properties and subproperties.

The property **hasDeviceCommand** is conceptualized as a **System** class performing a set of functions or commands over the set of devices:

- **hasDeviceCommand** (System hasDeviceCommand Device)
  - Domain: **System**
  - Range: **Device**

This property contains a series of subproperties, each corresponding to one type of functionality over the devices in the house:

- **SwitchOn** and **SwitchOff**, whose ranges are the classes **Fan**, **Heater**, **Lamp** and **Radio**
- **Open**, **Close**, **Raise** and **Lower**, whose range is the class **Blind**

Additional properties have been defined for the **Device** class, which are not related to the devices' functionality, such as:

- **hasColor** (Device hasColor Color)
  - Domain: **Device**
  - Range: **Color**
- **hasSize** (Device hasSize Size)
  - Domain: **Device**
  - Range: **Size**
- **locatedIn** a transitive function with four domains and ranges
  - Domain: **Device**, **SpecificLocation**, **Room**, **Area**
  - Range: **SpecificLocation**, **Room**, **Area**, **Home** (respectively)

The functionality considered for the telephone operator is described by the subproperties of the **hasTelephoneCommand** property. All of them take **System** as their Domain, and **DirectoryEntry** as their **Range**.

- **Call**
- **MakeConference**
- **Transfer**
- **CancelTransfer**
- **List** (entries)
- **Find** (entries)
- **Redial**

Additional properties have also been defined for the **DirectoryEntry** class which are not related to its functionality, such as:

- **hasEmail** (DirectoryEntry hasEmail Email)
  - Domain: DirectoryEntry
  - Range: Email

. . . and so on for each property: **hasFName**, **hasHName**, **hasHNumber**, **hasLName**, **hasNNumber**, **hasRelationship**, **hasSize**, **hasWNumber**.

Once it was determined that OWL would be the standard chosen, Prot´eg´e 3.1 open source ontology editor was chosen to help the development and reconfi guration of the ontology. Prot´eg´e is extensible and based on Java, and allows users to construct domain ontologies in various formats such as OWL, RDF, XML and HTML [Pro]. New devices will also be included.

2.3 Representing Domain Knowledge in the Linguamatics Interaction Manager

In this section we describe how domain knowledge is represented in the Linguamatics system. The Linguamatics Interaction Manager is an ontology-based dialogue system [MB03], where domain knowledge is described in terms of an ontology containing relationships between entities, and terminology (i.e. terms for concepts and synonyms). As an example we will describe the ontology which has been used in the demonstration version of the system. A similar, but slightly smaller ontology was used for the system installed at the Advantica Test House which was connected to a set of real devices including lights and blinds [CLAK05].

2.3.1 Representation of Home Devices

The Linguamatics ontology is similar to the Seville home ontology. The main differences are in the representation of functions such as **switch**, and in the use of more intermediate and higher level classes, often exploiting multiple inheritance.

device is parallel to Seville's **Device** class, but includes an extra layer of structure:

- **on-off device**
- **open-shut device**
- **sensor**

on-off device contains the classes:

- **cd**
- **light** (7 individuals)
- **television** (3 individuals)
- **radio**
- **washing machine**
- **cooker**
- **fridge**

- **freezer**

open-shut device contains

- **curtains** (5 individuals)
- **door**

sensor contains

- **smoke detector** (2 individuals)
- **carbon monoxide detector**

These subclasses of device are used for functional properties. For example, all **on-off devices** can be switched *on* or *off*. Descriptive properties are associated directly with individual devices. For example, rather than a subclass of **plasma tv** we associate the properties *lcd* or *plasma* directly with individual tvs. As well as the devices themselves, there are also classes for different states of devices. For example:

- **channel** (e.g. bbc1)
- **onoff** (on or off)
- **temperature** (0-40 degrees)
- **brightness** (0-100 percent)
- **track** (0-9)
- **volume** (0-100 percent)
- **openstate** (open or closed)

Devices are related to state classes using the **has-state** relationship. For example, **on-off device**s have the state **onoff**, **tv** have the state **channel** etc. Devices also inherit from their parent classes. For example, a **tv** has an **onoff** state inherited through being an **on-off device**.

2.3.2 Representation of the Home and the Location of Devices

The home has a top ontology concept of **location**. This has the children *upstairs*, *downstairs* and **rooms**. **rooms** includes subclasses for:

- **bathroom**
- **bedroom** (2 individuals)
- **hall**
- **kitchen**
- **landing**
- **lounge**

The relationship **in** is similar to Seville's **locatedIn** but has a single domain and range. The range is the class **location**. The domain is **item** which includes **location** and **device**.

2.3.3 Representation of Applications

The system includes an ontology of commands. This includes commands over devices e.g. **switch** or **close**, and commands which can be thought of as full applications or services e.g. **cinema booking** or **restaurant booking**. For example, **switch** takes three parameters, the device, the kind of state, and the state value. For example, to switch the channel for a tv with the unique identifi er, *tv-1*, to the channel *bbc1*, requires the command:

switch(*tv-1*, **channel**, *bbc1*)

The second parameter specifi es the class of state (in this case the state to change is in the class of **channel**). This is needed since the value itself may not be enough to indicate which state it is in. For example, volume and brightness both take a numeric value, so we need to distinguish between the following commands:

switch(*tv-1*, **volume**, *50*) **switch**(*tv-1*, **brightness**, *50*)

For cinema booking, the command requires the name of the fi lm, the time, and the number of people. For example:

cinema-booking(*Two Towers, 17.00, 3*)

Execution of commands is associated with effects (post conditions), and executing one command may cause a chain of commands to occur. Although there is no explicit representation of pre-conditions, it is possible to specify dependencies between parameters of a command (e.g. you can only switch a device to a state which is available for that device).

2.3.4 Representation of Menus

Even if a dialogue system is able to deal with fully specifi ed user commands (such as “switch the lounge tv to bbc1”), it can be useful to provide a menu-based scaffold, so that new users of a system can step through a series of options to get a better idea of the possibilities available. For example, in the Linguamatics system the menu-based scaffold allows users to navigate step-by-step to a particular service or device which they may not have known about a priori.

Ontological structure and menu structure are often similar. For example, a menu to access a particular device may ressemble the **is-a** structure of an ontology. For example, from a device menu a user may choose lights, heating, or sensors. From a lights menu the user may choose an individual light e.g. the lounge light. However, there is not always a one-to-one correspondence between the most convenient menu structure and the most natural ontology. For example the menu might only display the most commonly accessed devices, and put the rest in an **other devices** submenu, or not make these visible at all. In the Linguamatics home-control system, the natural ontology for the devices and the menu structure do diverge. For example,

1. device: the natural ontology clusters lights, washing machines etc. together as kinds of **device** whereas the menu structure only includes large devices such as the washing-machines, fridge and freezer under the **devices** node
2. lights: the natural ontology has all lights directly under a single node, **light**, whereas the menu structure has the most important lights directly under **lights**, plus an extra hierarchy of **more lights** for the less important lights.

3. room: the menu structure includes **room** under **home control** so that the user can navigate to particular devices by looking at the rooms in which the devices occur. The menu structure below **room** is determined by the **in** relationship rather than the **is-a** relationship.

Although the two structures diverge, the menu structure is assumed to be close enough to the natural ontology to be represented within the same is-a/in-a structure. The divergence is achieved partly by allowing multiple inheritance, and partly by providing a mechanism to allow nodes to be public (the menu structure) or private (the natural ontology).

The incorporation of menus into the ontology structure itself (as opposed to providing a separate tree structure) imposes the condition on the menu structure that subsumption relationships must be preserved: if two nodes N1 and N2 are in both the ontology structure and the menu structure and N1 is below N2 in the menu structure, N1 must be below N2 in the natural ontology structure. Note that N1 may be directly below N2 in one structure, and indirectly in another. This preservation of subsumption relationships gives the advantage that traversing a menu is a process of moving towards a more specifi c situation (a particular service, or a particular location in the house). This in turn means that answers which provide extra information (as defi ned by ontological subsumption relations) result in users jumping immediately to lower levels in the menu structure. It is an interesting question as to whether there may be cases where a well designed menu structure would need to depart further from the ontological structure. In the TAHI SA Project [CLAK05] this did not occur, even though the menu structure was designed by a separate team from the dialogue system designers.

2.4 Representing applications in GODIS

2.4.1 Applications, resources and resource types in GoDiS

In GoDiS, an application is implemented as a set of resources (domain knowledge in the form of ontologies and dialogue plans, device handlers, grammars etc.). Resources are regarded as objects of certain types (for example, `device_player` is an object of type `upnp_device`).

The TrindiKit defi nitions of resource interfaces sees them as abstract datatypes, i.e. specifi ed using relations, functions, and operations. Most resources (except `upnp_device`) in GoDiS are static, which means that there are no operations available for objects of these types. The basic GoDiS resource types are described in [Lar02a]. An example can be found in Section 2.4.4, which describes the GoDiS UPnP resource datatype.

2.4.2 Specifying a GoDiS application as an OWL ontology

In this section we show how to specify a complete GoDiS dialogue application as an ontology in OWL. This description can than be automatically transformed into working GoDiS resources. Given such a specifi cation, GoDiS applications can be modifi ed off-line using a standard OWL editor; new devices can be added, and the ontology can be extended (see also Section 5.3.3).

OWL, the Web Ontology Language, is a W3C standard for describing ontologies [W3C]. We are not using the full flavour of OWL, only the main components are needed. The main reason why we have chosen OWL instead of any other ontology description language is that it is a standard.

OWL ontologies

OWL has three main components – classes, individuals and properties:

Classes correspond to sets of individuals. The main relation between classes is entailment – that classes can be subclasses of other classes. There is no constraints on the subclass relation, which means that a class can e.g. be a subclass of several classes.

It is possible to form combined classes, most notably the intersection or the union of several classes. Classes can also be declared to be equivalent, or disjoint.

Individuals are class elements. Just as in set theory, an individual can be an element of several classes. This is equivalent to that the element is in the intersection class.

Properties correspond to relations between individuals. There are three kinds of properties – object, datatype and annotation properties. An object property has a domain and a range, which are themselves classes. The range of a datatype property is not a class, but instead a datatype, e.g. a number or a string. Annotation properties correspond to extra-logical properties, e.g. comments or version information.

Instead of writing an instance of a property as $P(a,b)$ we often say that "a has the property $P(b)$", or shorter $a :: P(b)$. That is, properties are seen as directed, from the domain to the range. Properties can also be declared to be functional, transitive, symmetric, et cetera. A property can also be a sub-property of another.

There is more to OWL than this, for example it is possible to create restrictions on classes and properties be logical formulae. There are three different levels of OWL – OWL Lite, OWL DL and OWL Full. Our ontologies are very simple, and fi t into OWL Lite, except for one detail: some of our properties have the class of classes as their range, meaning that in these cases ordinary classes act as individuals. This is only allowed in OWL Full, which is no serious problem since we are not interested in doing any reasoning on our ontologies.

Specifying a GoDiS dialogue application using OWL

In the rest of this section we will use an example ontology involving trams, ferries and buses in the Gothenburg area. This information is also used in the Gothenburg Tram Information System (GOTTIS).

The following domain-specifi c components have to be specifi ed to adapt GoDiS to a new domain:

- The *sorts*, the *sortal hierarchy*, and the *individuals* of each sort.
- The *actions* (from which requests are formed), and the *predicates* (from which propositions and questions are formed).
- The *sortal restrictions* on predicate parameters and *valid parameters* of predicates.
- The input and output *lexicon*, i.e. the domain-specifi c utterances.
- The *dialogue plans* for dealing with questions, requested actions, and the *device interface(s)* enabling interaction with external devices.

In the following these components are further described, together with their translation into OWL.

Previously all these components had to be described as predicates in Prolog (which is the language TrindiKit and GoDiS is programmed in). The problem with this approach is that many of the concepts have to be specified in several places, e.g. in different predicates. This is very error prone, since Prolog doesn't warn if a predicate or an atom is misspelled.

The solution is to specify the whole domain as an ontology, and then automatically generate the Prolog predicates. Since Prolog is an ontology specification language itself, this specification could already be done within Prolog; however, we have chosen to use a standardised description language. Some advantages with this is that it is possible to compare our solution with other ontology-based solutions, and that there are nice ontology editors making the editing simpler.

The main idea regarding how to specify a domain in OWL is to represent GoDiS concepts as OWL classes and/or properties:

GoDiS concept	**OWL toplevel classes**	**OWL properties**
sorts, individuals	Sort	
actions	Action	
predicates	Predicate	
sortal restrictions		*parameter* *valid*
domain-specific utterances	Syntax	*actionPhrase* *answerPhrase* *questionPhrase*
plans	Plan Construct	*construct* *precond*
devices	Device DeviceCommand	*device* *command*

The OWL classes and properties mentioned above are the main entities in a GoDiS domain description. We will explain the entities in more detail below.

Sorts, individuals and the sortal hierarchy

A *sort* is an OWL subclass of the top-level class Sort. An *individual* is an OWL individual of a certain sort. In GoDiS the sorts are declared with the predicate `sort/1`, and the individuals of a sort with the predicate `sem_sort/2`:

```
sort('Location').   sem_sort(gothenburg, 'Location').
sort('Stop').       sem_sort(central_station, 'TramStop').
sort('TramStop').   sem_sort(chalmers, 'TramStop').
sort('BusStop').    sem_sort(gibraltargatan, 'BusStop').
```

The Prolog database of sorts and individuals can be extracted from the OWL classes and individuals. The *sortal hierarchy* is an OWL subclass hierarchy. Consider the following example hierarchy from the Tram domain:

$$\text{Sort}\begin{cases}\text{Location}\begin{cases}\text{Street}\\ \text{Stop}\begin{cases}\text{TramStop}\\ \text{BusStop}\end{cases}\end{cases}\\ \text{Line}\begin{cases}\text{TramLine}\\ \text{BusLine}\end{cases}\\ \text{Route}\end{cases}$$

This subclass hierarchy says that each TramStop is also a Stop, which is also a Location. In GoDiS this is specified with the predicate `isa/2`, which can also be extracted from the ontology:

```
isa('TramStop', 'Stop').
isa('BusStop', 'Stop').
isa('Stop', 'Location').
isa('Street', 'Location').
```

Actions

The *actions* in GoDiS are used when the user requests the system to perform something, e.g. to restart the dialogue, or print the result on paper. These are also specified in GoDiS with the predicate `sem_sort/2`:

```
sem_sort(restart, action).
sem_sort(help, action).
```

In OWL, each GoDiS action becomes an individual in the top-level class Action, which gives us the following actions:

- restart ("start over please")
- help ("please give me help")

Predicates

The 1-place predicates in GoDiS are used for forming wh-questions, and for forming answers to questions. Each GoDiS predicate becomes an OWL individual in the top-level class Predicate. In the example domain there are the following predicates:

- departure ("from where do you want to go?" / "from the central station")
- destination ("to where do you want to go?" / "to the central station")
- shortest_route ("what is the shortest route between the two given stops?" / "the shortest route is …")

Sortal restrictions on predicates

Each GoDiS predicate has two domains:

Sortal restrictions are the sort(s) which the predicate can meaningfully be applied to, regardless of the specific capabilities of the domain application. Sortal restrictions encodes what the system can understand as being relevant in the domain, but possibly not accept and deal with. For example, the destination and departure predicates can be applied to locations in general. In GoDiS the sortal restrictions are declared by defining the Prolog predicate `sort_restr/1`:

```
sort_restr(destination(X)) :- sem_sort(X, 'Location').
sort_restr(departure(X)) :- sem_sort(X, 'Location').
```

In the ontology we handle sortal restrictions by the OWL property *parameter*, which associates each Predicate with a subclass of Sort:

destination :: *parameter*(Location)
departure :: *parameter*(Location)

Valid parameters are the sort(s) which are meaningful for the predicate in this particular domain, and which the system can accept and deal with. For example, the current system can only answer questions about the trams in Gothenburg, so only tram stops are valid whereas bus stops and other locations are not (although they are sortally correct). In GoDiS the valid parameters are declared by the predicate `valid_parameter/1`:

```
valid_parameter(destination(X)) :- sem_sort(X, 'TramStop').
valid_parameter(departure(X)) :- sem_sort(X, 'TramStop').
```

In the ontology we declare validity by the OWL property *valid*, in the same manner as above:

destination :: *valid*(TramStop)
departure :: *valid*(TramStop)

Since each valid parameter is also a parameter, the *valid* is a sub-property of *parameter*.

Domain-specific utterances

To the actions and predicates of a domain we associate utterances that the user and system can perform. In GoDiS these were originally defined in a Prolog predicate as a phrase spotting lexicon. However, in TALK deliverables D1.1 and D1.2 [LBC$^+$05, LAC$^+$06] we have shown how to implement the utterances as multilingual and multimodal grammars in Grammatical Framework (GF) [Ran04].

In the ontology we associate each utterance with a syntax tree from the GF resource grammar. The multilingual grammar rules are subclasses of the Syntax class, and the syntax trees can then be specified as individuals. Exactly how this is done is further described in TALK deliverable D1.5 [Pro06a].

For the purposes of this deliverable, it suffices to know that each utterance is associated with an individual in a subclass of **Syntax**. The domain-specific utterances all arise from the actions and predicates in the domain:

- For each Action there is an associated property *actionPhrase* which associates a verb phrase (VP) with the action.

- Each Predicate has two associated properties – a wh-question and an answer. Thus we have the two properties *questionPhrase* and *answerPhrase*, which associate a wh-question (WhQ) and an answer (Clause) to each predicate, .

In the GF resource grammar these categories (VP, WhQ and Clause) are very general, i.e. they can be realised in several different linguistic forms. This gives the freedom to e.g. either form a request from the user ("start over please"), or some kind of feedback from the system ("do you want to start over?", "I'm starting over").

Dialogue plans

The dialogue plans tells GoDiS how to solve the specifi c goals (actions or questions) that the user requests or asks for. For example, to be able to present the shortest route between two locations we fi rst have to calculate the route. And to be able to calculate the route, we have to know the departure and destination locations.

Plans are defi ned in GoDiS by the predicate `plan/2`, where the fi rst argument is the name of the plan, and the second is a list of *plan constructs*:

```
plan(find_route, [ findout(X^destination(X)),
                   findout(X^departure(X)),
                   dev_query(tram_db, X^shortest_route(X)),
                   dev_do(tram_gui, draw_route)
                 ]).
```

In the ontology we use the OWL classes Plan and Construct. To each Plan is associated a number of Constructs, by the property *construct*:

find_route :: *construct*(do_draw_route)
find_route :: *construct*(query_shortest_route)
find_route :: *construct*(findout_destination)
find_route :: *construct*(findout_departure)

However, since OWL properties are relations, there is no order between the constructs in a plan. An ordering is enforced by the property *precond* from Constructs to Constructs:

query_shortest_route :: *precond*(findout_departure)
query_shortest_route :: *precond*(findout_destination)
do_draw_route :: *precond*(query_shortest_route)

Note that there is no enforced order between *departure* and *destination*. This could potentially be used by the dialogue system to decide freely which question is the most natural to ask fi rst[1].

[1]GoDiS currently requires plans to be completely ordered, but allows flexibility regarding the order in the plan constructs are actually executed. Exploring the possible advantages of partially ordered plans is a topic for future research.

Each plan construct in GoDiS is a subclass of Construct:

$$\mathsf{Construct}\begin{cases}\mathsf{Findout}\\ \mathsf{DevQuery}\\ \mathsf{DevDo}\\ \ldots\end{cases}$$

To each of these subclasses there are some associated properties which determine the meaning of the construct. The Findout construct has a question to be answered, which is specified with the OWL property *predicate*:

> findout_departure :: *predicate*(departure)
> findout_destination :: *predicate*(destination)

Devices

Some plan constructs, e.g. the instances of DevQuery and DevDo, interact with a Device:

> do_draw_route :: *device*(tram_gui)
> query_shortest_route :: *device*(tram_solver)

Furthermore, each DevQuery construct asks its device a specific question, which can be specified by a Predicate:

> query_shortest_route :: *predicate*(shortest_route)

Finally, each DevDo construct has an associated DeviceCommand, which tells the device what to do:

> do_draw_route :: *command*(draw_route)

2.4.3 Menu-based dialogue plans

UGOT have developed techniques for building GoDiS applications by converting existing menu-based systems into dialogue systems [LCE01]. As part of TALK, we are extending this work to multimodal menu-based dialogue; this work is described in [MAB+05] and further in [Pro06b].

2.4.4 Connecting UPnP devices to GoDiS

In this section we describe briefly how GoDiS can interact with devices using a protocol based on UPnP (Universal Plug and Play) [2].

[2]This section is a modified version of material from [Lar02a], and is included here since it is relevant to the objectives of this deliverable.

We will mainly be dealing with devices that can be modelled as resources, i.e. that are *passive* (or *reactive*) in the sense that they cannot send out information unless queried by some other module. Of course, many devices are not passive in this sense but rather *active* (or *pro-active*), e.g. burglar alarms or robots[3].

To be able to hook up passive UPnP devices to GoDiS, we need the following:

1. device handler resources which communicate directly with the device itself; the device handlers can be said to represent the device in GoDiS;
2. a resource type for UPnP devices, specifying how devices may be accessed as objects of this type;
3. plan constructs for interacting with devices, and update rules for executing these plan constructs;
4. a Resource Interface Variable (RIV) to the TIS whose values are of the UPnP resource type; this variable hooks up devices to the TIS;
5. dialogue plans for interacting with devices.

In this section, we briefly describe the UPnP device handlers and the UPnP resource interface (including the resource type defi nition and the UPnP plan constructs) used by GoDiS. The `devices` RIV is described in Section 5.3. Dialogue plans for interacting with various UPnP devices are described in [Pro06b].

UPnP device handlers The device handler mediates communication between GoDiS and the device itself, and can be said to represent the device for GoDiS. We assume that each specifi c device has a unique ID, and is accessed via a separate device handler process. A device handler is built for a certain device type (e.g. the Panasonic NV-SD200 VCR), and each device of that type needs to be connected to a process running the device handler, in order to be accessed by GoDiS.

For UPnP devices, the device handler contains a specifi cation partly derivable from the UPnP specifi cations, but made readable for GoDiS (i.e. converted from XML to Prolog).

The device handler does the following:

- specifi es a set of actions and associated arguments
- specifi es a set of variables, their range of allowed values, and (optionally) their default value
- routines for setting and reading variables (`dev_set` and `dev_get`), for performing queries (`dev_query`), and for executing actions (`dev_do`)
- accesses the devicesimulation

[3]To handle active devices, we would need to build a TrindiKit module which could write information to a designated part of the information state based on signals from the device; this information could then trigger various processes in other modules. Still, even for an active device the solution we present here would be very useful; minimally, we would only need to add a module which sets a fhg in the information state whenever the device indicates that something needs to be taken care of, triggering other modules to query the device about exactly what has happened.

The UPnP resource interface In order to hook up a device to GoDiS one needs to define an abstract datatype for devices and declare a set of conditions and operations on that datatype. For GoDiS, we implement a generic resource interface in the form of an abstract datatype for UPnP devices.
In UPnP, a device is defined in terms of

- a set of variables
- a set of actions with optional arguments

In addition to getting the value of a variable, setting a variable to a new value, and issuing a command, we also add the option of defining *queries* to the device. These queries allow more complex conditions to be checked, e.g. whether two variables have the same value.
Based on this we define the datatype upnp_dev as in (1); here, *Var* is a device variable; *Val* is the value of a device variable, *Query* is a question, *Answer* is a proposition, α_{dev} is a device action, and *PropSet* is a set of propositions.

(1) $$\begin{array}{l} \textsc{type}: \text{upnp\_dev} \\ \textsc{rel}: \left\{ \begin{array}{l} \text{dev\_get}(Var, Val) \\ \text{dev\_query}(Query, Answer) \end{array} \right. \\ \textsc{opr}: \left\{ \begin{array}{l} \text{dev\_set}(Var, Val) \\ \text{dev\_do}(\alpha_{dev}, PropSet) \end{array} \right. \end{array}$$

Device actions may have one or more parameters; for example, in the VCR control domain there is an action *AddProgram* which takes parameters specifying date, program number, start time, and end time. The *PropSet* argument of `dev_do` is a set of propositions, some of which may serve as arguments to α_{dev}. In the resource interface definition, this set is searched by the device interface for arguments. This means that *PropSet* is not the exact set of arguments needed for α_{dev}; rather, it is a repository of potential arguments.
The relation between UPnP actions, device actions, and device operations is exemplified below:

- dev_do(my_vcr, AddProgram) is a UPnP action, which may appear in a plan
- AddProgram is a device action
- dev_do(AddProgram, {channel_to_store(1), start_time_to_store(13:45), ... }) is a device update operation

In addition to the datatype definition, one can define objects to be of that datatype. For each device that the system should recognise, the device ID should be declared to be of type `upnp_dev`.

2.5 Services in the Collaborative Problem Solving Framework

Our theory of dialogue as collaborative problem solving [BA05, Bla05] describes a general model of agent-agent collaborative problem solving (CPS) and then extends this to model human-agent and human-human dialogue as well. In addition to supporting more flexible dialogue (as described in Deliverable 2.1) and a wide range of dialogue behavior (as described in Deliverable 3.1), it also makes strides towards the development of truly domain-independent dialogue management. It does this through the use of an upper-level ontology of problem solving objects as well as the definition of an abstract level of additional information, both of which are then specialized and instantiated to the particular domains of interest.

In this section, we will first give an overview of the CPS model. We then discuss in more detail domain specialization of PS objects and how they are used to represent services in the dialogue system. Below in Section 3.2, we describe the dialogue manager itself.

As will become apparent in the text below, as opposed to typical representations of services, the CPS framework models more of what would be considered a hierarchical plan library. This, of course, is advantageous because it gives the dialogue system richer knowledge which allows collaboration at various levels of abstraction in the hierarchy (as described e.g., in [Car90]).

2.5.1 Overview of Collaborative Problem Solving Model

Our CPS model has been described in detail in the papers and deliverables mentioned above, and we will not attempt to give a full description here. Instead, we give an overview of the model from the standpoint of service representation and refer the reader to the other sources for more details.

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

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

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

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

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

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

Examples of Problem-Solving Behavior In order to better illustrate the problem solving behavior we want to cover in our model, we give several simple examples.

- *Prototypical*: Agent Q decides to go to the park (objective). It decides to take the 10:00 bus (recipe). It goes to the bus stop, gets on the bus and then gets off at the park (execution). It notices that it has accomplished its objective, and stops pursuing it (monitoring).
- *Interleaved Planning and Execution*: Agent Q decides to to go to the park. It decides to take a bus (partial recipe) and starts walking to the bus stop (partial execution) as it decides which bus it should take (continues to instantiate recipe)....
- *Replanning*: Agent Q decides to go to the park. It decides to walk (objective) and goes outside of the house (begins execution). It notices that it is raining and that doesn't want to walk to the park (monitoring). It decides instead to take the 10:00 bus (replanning)....
- *Abandoning Objective*: Agent Q decides to go to the park by taking the 10:00 bus. As it walks outside, it notices that it is snowing and decides it doesn't want to go to the park (abandons objective). It decides to watch TV instead (new objective)....

Problem-Solving Objects The CPS model operates on problem-solving (PS) objects. We defi ne an upper-level ontology of such objects, and defi ne the CPS model around them (which helps keep it domain-independent). The ontology can then be extended to concrete domains through inheritance and instantiation of the types defi ned here, as we will explain below.

The ontology defi nes six *abstract PS objects*, from which all other PS objects descend:

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

Recipe An agent's beliefs of how to attain an objective. Although we do not adhere to any specialized defi nition of recipe, one example is Carberry's domain plan library [Car90] which has action decomposition information about objectives. An agent's recipe library can be expanded or modifi ed through (collaborative or single-agent) planning.

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

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

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

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

Types in the model are defines as typed feature structures (similar to [PS94]), which allows us to track more information than a simple is-a ontology would allow.

For example, we include a RECIPE slot in the *objective* type, since part of the problem-solving process is the decision of which recipe to use to try to accomplish a chosen objective.[4] Likewise, the *recipe* type has a slot OBJECTIVES which records the set of objectives (i.e., subgoals or actions) which are part of that recipe. Domain-specific types can also add additional slots. In the MP3 domain, the type *play-song* inherits from *objective* and adds an additional HAS-SONG attribute which records the song to be played.

A possible approach, similar to that taken in most dialogue systems, is to have each of the slots described above directly take the value which was decided upon by the agents. Thus, when the agents decided on a recipe, it could be stored as the value of the RECIPE slot, and so forth. As will be discussed in Deliverable 3.1, however, this kind of representation is only able to represent the decisions made, but not the decision-making process itself. In true collaborative dialogue, such decision making can actually be the topic of multiple turns, including, for example, the proposal of possible values, evaluation of those values, and the final choosing of one.

In order to represent this decision-making information, we introduce two levels of indirection between each slot and its actual value, using objects we call *slots* and *fillers*. A full description of these is outside the scope of this deliverable. We refer readers to Deliverable 3.1 or [BA05] for details. Here we will just note that slots represent what possible values have been discussed and fillers wrap values (i.e., PS objects) and allow a space for recording context-specific evaluations which have been made about those objects in the decision-making process.

For the sake of completeness, we include in Figure 2.1 the formal feature structure type definitions for the upper-level PS object ontology, although space precludes an in-depth discussion of them (again, readers are referred to [BA05]). Type name and parent class are shown above the feature structure in the form **type** ← **parent**.

Domain Specialization The CPS model can be specialized to a domain by creating new types that inherit from the abstract PS objects and/or creating instantiations of them. This is described in more detail in Section 2.5.2 below.

The Collaborative Problem-Solving State The CPS state is part of the agents' common ground [Cla96], and models the agents' current problem-solving context. It is represented as an instance of type *c-situation* called the *actual-situation*. As the name implies, the *actual-situation* is a model of the agents' beliefs about the current situation and the actual problem-solving context.

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

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

Reasoning Act Families

[4]We give formal definitions of the types later.

$$\begin{bmatrix}\text{ID} & \textit{id}\end{bmatrix}$$

(a) **object ← ε**

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

(b) **slot ← object**

$$\begin{bmatrix}\text{CONSTRAINTS} & \textit{constraints-slot} \\ \text{IDENTIFIED} & \textit{set(filler}(\sigma)) \\ \text{ADOPTED} & \textit{filler}(\sigma)\end{bmatrix}$$

(c) **single-slot(σ) ← slot**

$$\begin{bmatrix}\text{EVALUATIONS} & \textit{evaluations-slot} \\ \text{VALUE} & \sigma\end{bmatrix}$$

(d) **filler(σ) ← object**

$$\begin{bmatrix}\text{IDENTIFIED} & \textit{set(filler(ps-object))} \\ \text{ADOPTED} & \textit{set(filler(ps-object))}\end{bmatrix}$$

(e) **multiple-slot ← slot**

$$\begin{bmatrix}\text{IDENTIFIED} & \textit{set(filler(constraint))} \\ \text{ADOPTED} & \textit{set(filler(constraint))}\end{bmatrix}$$

(f) **constraints-slot ← multiple-slot**

$$\begin{bmatrix}\text{CONSTRAINTS} & \textit{constraints-slot} \\ \text{IDENTIFIED} & \textit{set(filler(evaluation))} \\ \text{ADOPTED} & \textit{set(filler(evaluation))}\end{bmatrix}$$

(g) **evaluations-slot ← multiple-slot**

$$\begin{bmatrix}\text{CONSTRAINTS} & \textit{constraints-slot} \\ \text{IDENTIFIED} & \textit{set(filler(objective))} \\ \text{ADOPTED} & \textit{set(filler(objective))} \\ \text{SELECTED} & \textit{set(filler(objective))} \\ \text{RELEASED} & \textit{set(filler(objective))}\end{bmatrix}$$

(h) **objectives-slot ← multiple-slot**

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

(i) **ps-object ← object**

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

(j) **objective ← ps-object**

$$\begin{bmatrix}\text{ACTIONS} & \textit{objectives-slot} \\ \text{ACTION-CONSTRAINTS} & \textit{constraints-slot}\end{bmatrix}$$

(k) **recipe ← ps-object**

$$\begin{bmatrix}\text{ACTUAL-OBJECT} & \textit{id}\end{bmatrix}$$

(l) **resource ← ps-object**

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

(m) **constraint ← ps-object**

$$\begin{bmatrix}\text{ASSESSMENT} & \textit{unstructured}\end{bmatrix}$$

(n) **evaluation ← ps-object**

Figure 2.1: PS Object Definitions (1)

$$\begin{bmatrix} \text{PENDING-PS-OBJECTS} & \textit{set(ps-object)} \\ \text{PS-OBJECTS} & \textit{set(ps-object)} \\ \text{PS-HISTORY} & \textit{list(interaction-act)} \\ \text{FOCUS} & \textit{stack(object)} \\ \text{OBJECTIVES} & \textit{objectives-slot} \end{bmatrix}$$

(a) **c-situation ← ps-object**

Figure 2.2: PS Object Definitions (2)

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

Commitment Act Families

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

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

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

Grounding Acts The model as it stands thus far makes the simplifying assumption that utterances are always correctly heard by the hearer and that he also correctly interprets them (i.e., properly recovers the intended (instantiated) interaction acts). In human communication, mishearing and misunderstanding can be the rule, rather than the exception. Because of this, both speaker and hearer need to *collaboratively* determine the meaning of an utterance through a process termed *grounding* [Cla96].

We add grounding to our model by utilizing the *Grounding Acts* (GAs) proposed as part of Conversation Acts theory [TH92][5] and used in Traum's computational model of grounding [Tra94]. In our model, we expand the definition of GAs to allow them to take individual IntActs as arguments. As the grounding acts themselves are not our focus here, we discuss them only briefly. The Grounding Acts are as follows:

Initiate The initial contribution of an IntAct.

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

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

Repair Changes some part of the IntAct.

ReqRepair A request that the other agent repair the IntAct.

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

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

In the model, an IntAct is not successfully executed until it has been successfully grounded. This is typically after an *acknowledge*, although see [Tra94] for details.

2.5.2 Domain Specialization of Objects

Now that we have given an overview of the model, we describe in more depth how PS objects are specialized to a given domain.

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

All PS object types (including new types created by inheritance) can be further specialized by instantiation, i.e., by assigning values to some set of their attributes. This can be done both at design time as we discuss here and it happens at runtime as part of the problem-solving process itself (see discussion in Deliverable 2.1).

We now describe specialization for components of service descriptions: *resources* and *objectives*.

Resources Although modeling resources is not part of modeling services per se, services typically work on certain resources, so we include their definition here.

New types of resources can be created by inheriting from the *resource* type. Figure 2.3 shows a partial definition of a *song* object from our MP3 domain. Here the individual traits are simply added as slots which take type *sslot* as values.

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

$$\begin{array}{c}\boldsymbol{song} \leftarrow \boldsymbol{resource} \\ \begin{bmatrix} \text{HAS-ALBUM} & \textit{sslot(album)} \\ \text{HAS-ARTIST} & \textit{sslot(artist)} \\ \text{HAS-LENGTH} & \textit{sslot(time)} \\ \ldots & \end{bmatrix}\end{array}$$

Figure 2.3: Definition of *song*

$$\begin{array}{c}\boldsymbol{play\text{-}song} \leftarrow \boldsymbol{objective} \\ \begin{bmatrix} \text{HAS-SONG} & \textit{sslot(song)} \end{bmatrix}\end{array}$$

Figure 2.4: Definition of *play-song*

Objectives As discussed above, objectives are considered both goals and actions in the CPS model, and as such, they are what we use to model services.

Here, new types of *objectives* are created in a similar way to creating new *resources*. A new *objective* inherits from the *objective* type and adds certain new slots. In the case of *objectives*, these new slots can be seen as recording the *parameters* of the action. For example, in Figure 2.4, we show the defi nition of the *play-song* objective in our MP3 domain. This objective adds a new HAS-SONG slot, which is used to record problem-solving about the particular *song* that is to be played.

Chapter 3

Dialogue Management to Support Dynamic Reconfiguration

3.1 Introduction

This chapter considers some of the implications of reconfigurability on dialogue management. The first section concentrates on the CPS system where work has been undertaken to achieve greater reconfigurability. The second section describes the GoDiS approach which combines domain independent update rules with domain-specific resources. Although the GoDiS system and the Linguamatics system were designed from the start to be domain independent, there is a danger of pushing application specific knowledge into ever more expressive specification languages. In the extreme, the dialogue system becomes an operating system, and the application specifications become programs. The final section considers a different approach of allowing more flexible control between dialogue and task managers so that general action management is taken back outside the dialogue managers responsibility.

3.2 Towards Domain-Independent CPS-based Dialogue Management

In this section, we describe our work towards creating a domain-independent dialogue manager based on our model of collaborative problems solving (described in Section 2.5). We believe that domain-independent dialogue management is important for two interrelated problems for dialogue systems: portability and dynamic reconfiguration. Portability is helped here because, if a truly domain-independent dialogue manager can be produced, it would mean that the dialogue manager itself (in ISU, the update rules) would not need to be changed to support new domains. Similarly, for dynamic reconfiguration, a new device or service could be added to the system without the need to modify the dialogue manager.

It is our hope that the CPS model of dialogue sufficiently abstracts dialogue in such a way that the same set of CPS-based update rules could be used for different domains. As the title of this section suggests, we do not yet claim to have a domain-independent CPS-based dialogue manager, although we think we have made considerable progress towards this end.

In the rest of this section, we will first describe our dialogue manager, and how we attempt to make it domain independent using abstraction in the PS object hierarchy. In the process of building this dialogue manager, we also discovered some types of domain-specific knowledge *outside* the CPS model proper, which are also necessary for the dialogue manager. This is described as well, and then we describe parts of the dialogue manager which are still domain specific. Finally, we discuss future directions for this work.

3.2.1 The SAMMIE Dialogue Manager

As outlined in Deliverable 5.2, we have built a dialogue manager for the SAMMIE dialogue system which supports a subset of the CPS model discussed above. The ideal situation for exploring domain independence would be to show the same dialogue manager working in many different domains. Unfortunately, as the SAMMIE system only supports the MP3 domain thus far, this has not been a feasible option. Instead, we base our claims of domain independence on the weaker evidence of update rules only accessing information through general mechanisms at the upper-ontology level of the CPS model. This is necessary, but not sufficient proof of domain independence. While it is able to show that individual rule are, in some sense, domain independent, it cannot show that the *set* of rules is sufficient for all domains. We are convinced that our current set of rules is *not* sufficient for all domains — however, it is our hope that moving to new domains would be more a process of rule *augmentation* than rule *modification*.

As described in Deliverable 2.1 [MAB$^+$05], dialogue management in the SAMMIE system can be grouped into three separate processes:

Integrating Utterance Information Here the system integrates grounding acts — by both user and system — as they are executed.

Agent-based Control Once executed grounding acts have been integrated into the dialogue model, the system must decide what to do and what to say next.

Package and Output Communicative Intentions During the first two phases, communicative intentions (i.e., instantiated grounding acts) are generated, which the system wants to execute. This last phase packages these and sends them to the generation subsystem for realization. When realization is successful, the information state is updated using the rules from the first phase.

In the dialogue manager, the first and last phases are handled entirely by rules that refer to only the upper-level of the CPS ontology (e.g., *objectives, resources*, and so forth). For example, one particular rule fires when an act in the `c-identify` family has been generated (by the generation of a `complete` act that wraps it). The rule takes the PS object parameter of the `c-identify` and inserts it into the IDENTIFIED set found within the context variable of the `c-identify`, regardless if it is a *play-song* objective, a *song* itself, or some other PS object from a completely different domain.

Rules in these phases handle the integration semantics of the CPS acts themselves and are described in more detail in Deliverable 3.1. These are the core of the CPS model, and they would only change if the CPS model itself changed. We do not believe these would need to change to support new domains or services.

The middle level (*agent-based control*) is, however, where reconfigurability can become an issue. It is here where the dialogue manager makes decisions about what to do and say next. In our dialogue manager, we were able to formulate many of these rules such that they only access information at the upper ontology level, and do not directly access domain-specific information. As an example, we illustrate a few of these here.

System Identifies a Resource by Request The following rule is used identify a resource in response to a request by the user:

- **if**
  - `c-identify-resource` is being negotiated, and
  - the ACTUAL-OBJECT slot of the *resource* is blank (i.e., this is a request that the system identify the resource), and
  - the system can uniquely identify such a resource given the *constraints* used to describe it
- **then**
  - add the found *resource* to the ACTUAL-OBJECT slot
  - create a new `continue` of the `c-identify-resource`
  - add this to the queue of responses to be `initialized`

As can be seen, this rule relies only on the (abstract) information of from the CPS model — namely, that all *resources* which have been definitely referred to should have a pointer in the ACTUAL-OBJECT slot to the value in the system's knowledge base (see discussion in Deliverable 3.1). When this is not the case, and this *resource* is part of a pending `c-identify-resource`, then this means the user has made a request that the system identify such a *resource* (given constraints the user may have provided). In the MP3 domain, this rule is used to provide user-requested sets of information from the database (e.g., in response to "Which Beatles albums do you have?"). No domain-specific knowledge is encoded in this rule.

System Prepares an Objective to be Executed The following rule is used when the system marks a top-level objective to be executed next. Note that the current version of the system does not support hierarchical plans, thus the assumption is that this is an atomic action. Also, the system currently assumes that atomic action execution is instantaneous — thus nothing will stay in a SELECTED set once execution has begun.

- **if**
  - An *objective* is in the SELECTED set in the OBJECTIVES slot in the CPS state
- **then**
  - Put the *objective* on a system-internal stack to signal that execution should begin.

This is an example of a simple rule which prepares an *objective* for execution. Similar to the rule just described, no domain-specific information is necessary here — all *objectives* are handled the same, no matter from which domain.

Although we were able to formulate many rules with information available in the CPS model, we encountered some which needed additional information from the domain — including the case where the atomic action execution should actually take place. We now turn our attention to these cases.

3.2.2 Abstracting Additional Domain Information

In the rules discussed above, simple knowledge implicit in the use of abstract PS objects was suffi cient for encoding rules. However, there were a few cases which required more information. In this section, we discuss those cases for which we were able to fi nd a solution in order to keep the rules domain-independent. In the next section, we discuss rules which needed to remain domain-specifi c, and the reasons for that.

Just because domain information is needed for rules does not mean that we cannot write domain-independent rules to handle them. What is required, however, is the specifi cation of an abstraction for this information, which every new domain is then required to provide.

In the MP3 domain, we have identifi ed two general types of this kind of knowledge. We do not consider these to be a closed list:

Execution Knowledge One of the example rules above showed how the decision to begin execution of an atomic action is made. However, the *actual* execution requires knowledge about the domain which is not present in the CPS model.

In the current system, a domain encodes this information in what we call a *grounded-recipe*, which we have provisionally added as a subtype of *recipe*. A *grounded-recipe* contains a reference to the *objective* it fulfi ls as well as a pointer to a Java class which implements the abstract (Java) class `GroundedRecipe`. This abstract class provides an abstract method to be overridden in which Java code corresponding to the execution of the objective can be written.

- **if**
  - an *objective* has been chosen for execution
- **then**
  - look up a matching *grounded-recipe* for the *objective*
  - execute the `execute` method of the Java class pointed to in the *grounded-recipe* (passing in the *objective* itself as a parameter).

This abstraction supports a rule which looks up a corresponding *grounded-recipe*, and then calls the appropriate Java method in the class associated with it, keeping the execution rules domain-independent.

Evaluation of PS Objects A more general issues which surfaced was the need to make evaluations of various PS objects in order to decide the system's acceptance/rejection of them within a certain context. Although we believe there is a need to specify some sort of general classifi cation for these, only one such evaluation came up in the MP3 domain.

In deciding whether or not to accept the identifi cation of a fully-specifi ed *objective*, the system needed a way of checking the preconditions of the *objective* in order to detect potential errors. For example, the SAMMIE system supports the deletion of a song from a playlist. Now, grounding-level rules (not detailed here) take care of defi nite reference errors (e.g., mention of a playlist that does not exist). However, if reference to both objects (the song to be deleted and the playlist) is properly resolved, it is still possible that the user asked to delete a song from a playlist when the song is not actually on the playlist. Thus, we needed a way of checking this precondition (i.e., does the song exist on the playlist). Similarly, we needed a way of checking to see if the user request playback of an empty playlist (i.e., a playlist that does not contain any songs).

As a simple solution, the dialogue manager uses an abstract interface to allow rules to check conditions of any objective:[1]

- **if**
  - a `c-identify-objective` is pending for a fully-specifi ed *objective*, and
  - `CheckPreconditions` fails for the *objective*
- **then**
  - add a `reject` of the `c-identify-resource` to the output queue.

3.2.3 Domain-specifi c Rules in the System

Despite our best efforts, a few domain-specifi c update rules are still present in the dialogue manager. We describe two of these here which are used to cover holes which the CPS model does not yet (adequately) address. We hope to expand the model in the future so that these rules can also be generalized.

In the MP3 domain, we support the creation of (regular) playlists as well as so-called auto-playlists (playlists created randomly given constraints). Both of these services correspond to atomic actions in our domain and would be theoretically handled by some of the rules for execution described above. However, these are both actions which actually return a value (i.e., the newly-created playlist). This kind of return value is not supported by the representation currently used in the CPS model. For this reason, we support the execution of both of these actions with special rules.

3.2.4 Discussion

In this section, we have presented our CPS-based dialogue manager, and shown how, for the most part, we have been able to keep individual rules domain-independent. Although much work needs to be done to achieve true domain-independence, we believe our work has shown that the CPS model may be viable for creating such a domain-independent dialogue manager.

3.3 Domain independent update rules in GODIS

GoDiS is modular in the sense that the Dialogue Move Engine itself is domain-independent, and all domain-dependent and application-specifi c knowledge is located in the resources (typically grammar, language model, domain knowledge and device interface). See Figure 3.1 for an example.

[1]This is currently implemented just as a single Java method with an `if-else` statement. For better reconfi gurability, this could be sent to an external agent which could register this service for objectives in different domains.

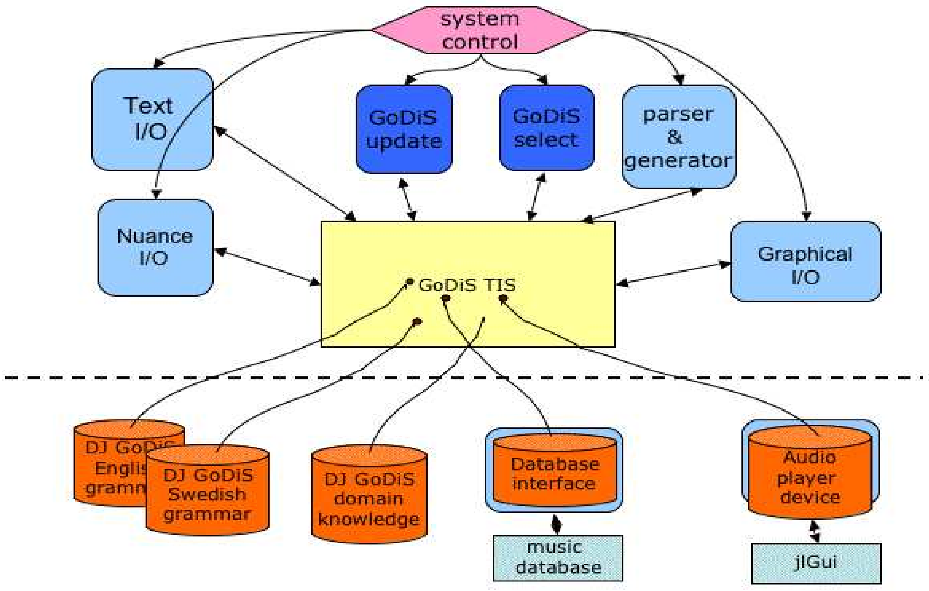

Figure 3.1: Example of domain-independent modules (above the dotted line) and domain-specific resources (below the line) in the DJ-GoDiS application

3.3.1 Resources and Resource Interface Variables in TrindiKit

In any TrindiKit system, resources are connected to the Total Information State (TIS) via Resource Interface Variables (RIVs). There is one RIV for each resource (e.g. domain knowledge, database, grammar), and the values of RIVs are names of particular application-specific resources (e.g. `domain_player`, `domain_lights`, `device_player`, `grammar_player_english`).

```
gf_grammar = gfgrammar_player_english
domain = domain_player
devices = [ player = device_player, dbase = device_dbase ]
```

Each RIV is declared to be of a certain type (e.g. `domain`, `upnp_device`, `nuance_grammar`) and the resources are regarded as objects of these types (for example, `device_player` is an object of type `upnp_device`). The definition of a resource type declares the methods (functions, relations, operations) available for objects of that type (see also Section 2.4.1).

```
gf_grammar : gfgrammarType
domain : domainType
devices : record([L1:T1, ..., Lm:Tn]) where Tn=upnp_deviceType
```

3.3.2 Accessing resources from update rules

Update rules communicate with resources by looking up the value of RIVs; this is done by prefixing the name of an RIV with the TrindiKit evaluation operator ("$"). For example, to check whether an answer `A` is relevant to a question `Q` (which depends on the current domain), the following condition is included in a rule:

```
$domain :: relevant_answer(A, Q)
```

If the `domain` variable has the value `domain_player`, this condition will is equivalent to the following:

```
domain_player :: relevant_answer(A, Q)
```

In this way, it is possible to make TrindiKit update rules application independent, as long as each application resource defines the appropriate methods, as declared by the corresponding datatype definition.

3.3.3 Switching resources

To switch to a new application, one simply has update rules change the values of the RIVs to point to the resources for the desired application. Since the update rules access the resources via these variables, the rules themselves do not need to be changed. This greatly simplifies the implementation of plug-and-play facilities in GoDiS, as explained in Section 5.3. The GoDiS DME is described in detail in [Lar02a].

In some cases it is impractical to have modules inspecting the TIS to access resources (typically, for modules which are separate OAA agents). Instead, the module itself loads the resource directly instead of accessing it via the TIS and the RIVs. In this case, it is possible to set up triggers that fire whenever the value of a certain RIV changes, and that broadcasts an OAA solvable that tells the module to load the new resource.

3.4 Mixing control between the Linguamatics Interaction Manager and external tasks

So far, when we have considered reconfiguring for a new task we have still taken a relatively static view of the relationship between the dialogue manager and any external processes or tasks. Although new devices or services are plugged in, the dialogue manager is still very much in control of the interaction with the user.

However, a dynamically reconfigurable system can interact in a much more flexible way. For example, a task may want to interact with a user, and can send the dialogue manager a goal to be satisfied plus an ontology fragment. In the extreme, the task can control each step of the dialogue, asking the dialogue manager to do a single step at a time.

In the Linguamatics system, the recommended interaction between task and dialogue management is to allow the dialogue manager to do what it is best at i.e. to communicate with the user. Thus, if a task needs five pieces of information in no particular order, it should pass on this request as a whole to the dialogue manager, rather than requesting the first piece of information, then the second etc. This allows the dialogue manager to reorder questions, or respond to the user answering more than one request at a time (see the discussion on coherence in D2.1).

To experiment with joint control between an external application and the dialogue manager, we integrated the Linguamatics Interaction Manager with two prototype applications developed by the University of Loughborough. The first is based on the FoodWare recipe system. The application monitors food coming in and out of the house using RFID tags, and proposes a selection of recipes to a user based on their food preferences. To achieve this, the application passes the dialogue manager a task to find out which recipe the user would prefer, along with a mini-ontology consisting of possible recipes and their properties. The second application uses RFID tags to monitor people's movement around a house. The application resets the dialogue manager's default context according to where the person is in the house (assuming the user is not already in the middle of a dialogue). This is equivalent to traversing the menu tree to get to that location. For example, if the user moves into the kitchen they can then say "turn on the light" rather than "turn on the kitchen light" since it is assumed that they are in the context of the kitchen. Integration of both applications went smoothly suggesting that joint control models are worth exploring. Given the success of this prototype, we are investigating further separation of task and dialogue management. Consider, for example, testing of preconditions. The Linguamatics system has a notion of valid options (e.g. the film Two Towers at 5pm vs. the film Oliver Twist at 4pm). If a user asks for "Two Towers at 4pm" the response is to ask "did you mean Two Towers at 5pm or Oliver Twist at 4pm". However, evaluation of preconditions could be arbitrarily complicated, especially if combined with warnings for unlikely combinations of parameters. Using the separation of task management and dialogue management there are now two potential ways of dealing with this, without having to evaluate preconditions within the dialogue manager:

1. The dialogue manager communicates its results back to the task manager after every user utterance. This gives the task manager a chance to check if any preconditions are failing. The dialogue manager can still be given multiple items of information to request at once, so the interaction with the user should be unaffected.

2. The dialogue manager controls the gathering of multiple pieces of information, but opens up a second communication with the task manager to ask for current results to be checked after receiving

each new constraint.

The general approach is therefore to treat the dialogue manager solely as a communication agent. Information about the tasks and services used by the dialogue manager does not include workflow or process elements, thereby avoiding the need for the dialogue manager to act as a proxy operating system.

Chapter 4

Dealing with Multiple Applications

4.1 Introduction

In the introduction we discussed how the user gets benefi t from having commands across multiple devices. The same can be true of different services. The aim here is to see what can be done when two individual services are plugged separately into the system. This can be contrasted with cases where a system designer specifi cally codes for the interaction between two services (effectively defi ning a third service).

This section introduces some basic distinctions regarding various aspects of plug-and-play. We sketch a framework for classifying implementations of plug-and-play behaviour in dialogue systems along several conceptually independent dimensions.

4.1.1 Device and application (service) plug-and-play

In this section, we introduce some basic terminology, and distinguish between different kinds of reconfi gurability concerning devices and applications.

Devices and device modification

By *device*, we mean something close to the everyday meaning of "device" but perhaps slightly more general. Examples of devices are: a lamp, an MP3 player, an agenda, a toaster, a clock but also e.g. a music database.

A very basic type of reconfi guration is adding new functionalities to a device, or altering ontological or lexical knowledge to an existing device. We refer this as *device modification* and regard it as a different kind of reconfi gurability than what is typically meant by "plug-and-play".

Device plug-and-play

Devices can often be grouped (more or less naturally) into *device clusters*, e.g. the set of lamps in a home, the set of simple electrical appliances in a home (including lamps), or an mp3 player plus a song database.

By an *application* (or *service*), we mean a dialogue interface for communicating with a device or a device cluster. Within an application, all devices are on an equal footing in the sense that all devices are equally accessible to the user. An application is typically designed to cater for a cluster of related devices, independently of any other applications that might run at the same time.

Typically, the choice of which devices to group together in a cluster is decided by the application designer or possibly the user. The motivation for clustering certain devices together is in a sense arbitrary but can be motivated by aspects both of implementation and of usability. For example, it seems not very useful to cluster a kitchen lamp and a music database into a single application.

The consequences of device clustering on a dialogue interface depend on the kind of plug-and-play that the interface implements. It should be noted that for some types of plug-and-play, the concept of an application as a device cluster is not very useful; in this case one may regard the interface as a single-application interface.

By *device plug-and-play*, we simply mean the ability to connect new devices to an application. An application is dynamic (with respect to plug-and-play) to the extent that new devices can be added (and subtracted).

Application plug-and-play and application modification

On the next level we have the possibility of running several applications simultaneously, and consequently a concept of *application plug-and-play*, i.e. the ability to connect new applications to a dialogue interface. The devices available in an application *A* may not be directly available in a different application *B* running simultaneously, and controlling them typically requires switching to application *A*. We distinguish between the set of *connected applications* - all applications that are available through an interface - and the *active application* (or applications) whose devices are currently available directly.

Application plug-and-play typically (but perhaps not necessarily; this depends on the level of reconfi gurability of the dialogue system in question) requires that the applications are based on the same *dialogue system configuration*; by this we mean a set of modules (or agents) such as a dialogue manager, speech recogniser, interpretation engine, etc., as well as algorithms to coordinate these modules. A dialogue system is dynamic (with respect to plug-and-play) to the extent that new applications can be added (and subtracted).

By analogy with “device modifi cation” we can also talk about *application modification*, by which we mean “soft” modifi cations of functionalities, ontological and lexical knowledge (e.g. changes regarding the location and descriptors of devices in an application). Modifi cation of devices in an application entails application modifi cation, since the ontological knowledge related to an application includes the ontological knowledge of the devices included in the device cluster for the application.

4.1.2 Weak and strong plug-and-play

By *weak* (or *token-level*) plug-and-play we mean adding a new device or application of a known type, i.e. of the same type as an already connected device or application. In this case it is possible to re-use e.g. ontological knowledge which is already available

In the case of *strong* (or *type-level*) plug-and-play, a device or application of a new type is added. This requires making new ontological, lexical and other knowledge available to the interface.[1]

[1]“Soft plug-and-play” can now be analysed as one of device modifi cation, application modifi cation, or weak device plug-and-play.

4.1.3 Off-line and on-line plug-and-play

In this section, we describe three basic ways in which an interface can be set up to allow for plug-and-play behaviour.

Pre-programmed applications/devices

On this approach (which is arguably not plug-and-play in any strict sense), applications are pre-programmed by hand to know about a fixed (static) set of possible simultaneous applications. To add a new application to a cluster, all applications in the cluster need to be modified by hand.

To add a new device to an application, the application is modified by hand to deal with a new device.

Off-Line plug and play

To plug in a new application in the case of off-line application plug-and-play, the following steps are required:

- the dialogue system is halted
- the application cluster is extended by adding one or several new applications to a list of applications to be run simultaneously
- all applications (more or less automatically) exchange the appropriate information; the information to be exchanged depends on the chosen application switching behaviour (e.g. indirect switching)
- the dialogue system is restarted

In this case, we have a “semi-dynamic” application cluster.

For on-line *device* plug-and-play, the following steps are required:

- the dialogue system is halted
- the application is extended by adding one or more new devices to a list (or similar) of devices to be handled by the application
- the dialogue system is restarted

On-line plug and play

For on-line application plug-and-play, applications can exchange the appropriate information for enabling application switching on-line, i.e. without requiring stopping and restarting the system. In this case, we have a fully dynamic application cluster.

Similarly, for on-line device plug-and-play, devices can be added to an application while the application is running. The appropriate information will be broadcast from the device (or some information storage related to the device).

4.1.4 Application switching strategies

Strategies for handling multiple applications can be classifi ed according to how the user is able to switch to a different application. We assume that there is a cluster of *connected applications* and that one of these is the *active device*[2]. The defi ning feature of the active application is that all its' devices are maximally available and can be manipulated with as little effort as the dialogue system allows. Manipulation of devices connected to non-active applications may require more effort (typically, switching to the new application).

Note that there is no need for general "device switching" strategies apart from those implemented in the application (e.g. in the form of a menu system to access different devices).

We distinguish three strategies for application switching: (explicit) indirect switching, explicit direct switching, and implicit direct switching.

(Explicit) indirect switching

In this case, the user can switch fi rst to a "meta-application" (e.g. by saying "Change domain") and then to a parallel application ("the radio please"). This option requires only that each application offers the possibility of switching to the meta-application. Applications do not need to know about other applications, apart from the meta-application. The dialogue manager must enable switching to the meta-application and from the meta-application to any other application.

Example:

```
U: Turn on the kitchen lights.
S: OK.
S: Switch application
U: OK. Switching to meta application.
S: Which application do you want?
U: mp3 player
S: mp3 player here
```

Depending on the choice of plug-and-play option from the three described above, the following application capabilities are required:

- **Pre-programmed applications**: Meta-application is pre-programmed to know about all connected applications
- **Off-Line plug and play**: New applications can be added to meta-application off-line
- **On-line plug and play**: New applications can be added to meta-application on-line.

Explicit direct switching

The user can explicitly switch directly to another application by issuing a request including the name of the application (e.g. "Go to the mp3 player"). The dialogue manager must allow switching to any other

[2]In some cases it may be useful to have several devices active at the same time.

connected application. Each application grammar must contain the names of all other applications, e.g. as complements of “Switch to..”. No other grammar modifi cations are needed.
Example:

```
U: Turn on the kitchen lights.
S: OK.
U: Switch to the mp3 player
S: OK. Mp3 player here.
U: Add a song to the playlist
```

Depending on the choice of plug-and-play option from the three described above, the following application capabilities are required:

- **Pre-programmed applications**: Each application grammar contains names of other applications
- **Off-Line plug and play**: Applications can be updated off-line to learn about new applications, by adding new application names to grammars
- **On-line plug and play**: Applications can be updated on-line to learn about new applications, by adding new application names to grammars

Implicit direct switching

The user can switch to another application by addressing it directly (e.g. “Add a song to the playlist”). In this case, the dialogue manager must allow addressing non-active application, and each application grammar must contain subsets of all other application grammars (e.g. user commands and questions).
Example:

```
U: Turn on the kitchen lights.
S: OK.
U: Add a song to the playlist
```

Depending on the choice of plug-and-play option from the three described above, the following application capabilities are required:

- **Pre-programmed applications**: Each application grammar contains subsets of other application grammars
- **Off-Line plug and play**: All applications can be updated off-line to learn about new applications, by adding a subset of a new application grammar to all application grammars that will be connected simultaneously
- **On-line plug and play**: All applications can be updated on-line to learn about new applications, by adding subset of a new application grammar to all connected application grammars

4.2 Application accommodation in GoDiS

This section describes the application selection strategy implemented in GoDiS. The plug-and-play strategies and solutions are described in Section 5.3.

The application switching strategy we will focus on here is the most powerful one, *implicit direct application* switching according to the taxonomy in Section 4.1 This strategy, which we will also refer to as *application accommodation*, requires fi guring out which device the user is addressing in the absence of explicit mention of a specifi c device.

Other useful strategies described below include addressing several applications in one utterance (as in ”Turn on the kitchen lamp and play the radio”)

4.2.1 Dialogue management for application accommodation

The basic strategy for dealing with the dialogue management aspect of application switching in GoDiS is to update the resource interface variables (RIVs; see [SIR02]) so that the active application is always the one that is accessed when update rules consult the resources.

By analogy with GoDiS' capabilities for question accommodation [Lar02b], we can think of implicit indirect application switching as a kind of “application accommodation”. This means that rather than having to explicitly request a different application than the currently active one, the system will fi gure out which application the user is addressing and adapt so that this application becomes active. This will enable the system to load the appropriate plan from the domain knowledge resource for the application, and to integrate the request or question that triggered the application accommodation.

The grammar for any application will include parts of the grammars of the other connected applications. The parser will provide output indicating, for each interpreted dialogue move, which application grammar was used to parse that move.

In GoDiS, it is possible for the user to take initiative at any time and introduce a new task (action or issue). We want to allow for this behaviour also for tasks belonging to separate applications, i.e., we want to allow for *multiple simultaneous tasks across applications*. In analogy with the GoDiS strategy for keeping track of simultaneous tasks by storing them on a stack and dealing with the topmost one fi rst, we will keep track of a stack of activated applications, where the topmost application is the currently active one, i.e. the one whose devices are maximally available.

Extending the Information State for application accommodation

We fi rst add a GoDiS datatype `application` to be able to represent applications in the TIS.

Second, we include a TIS variable `all_devices` whose value is a record of all connected devices for each connected application.

Third, we add add TIS variable `active_apps` of type `stack(application)` to keep track of the activated applications. The topmost application is the currently active one.

We also add a TIS variable `app_per_move` whose value is a so-called association set which allows looking up the application for any dialogue move parsed from the latest user utterance and (as stored in the `latest_move` variable).

For example, if the currently active application controls lights (GoDiS-deLux), and the user says “Add a song to the playlist”, this can be parsed by the part of the global grammar that is derived from the mp3 player application (DJ-GoDiS). The GF parser output will thus indicate that this move belongs to this application (called `player` below). The parser module will update the TIS to include the following:

```
latest_moves = { request(playlist_add) }
app_per_move = { < request(playlist_add), player > }
```

IS update strategies for application switching

This section describes the information state updates associated with application switching in GoDiS. For the simple case, we provide a walk-through of the implementation; for the more complex cases we give a brief description and refer the interested reader to the code and associated documentation.

Switching to a non-activated application To make sure the application which is addressed by a move is also the active one, the GoDiS DME tries the following rule before integrating each move:

```
rule( switchApplicationImplicit,
   [ fst( $/private/nim, Move ),                               1
     not empty($app_per_move),                                 2
     AddressedApp = $$assoc($app_per_move, Move),              3
     not in($active_apps, AddressedApp) ],                     4
   [ domain := $$dash2underscore(domain-AddressedApp),         5
     Language = $language,                                     6
     gf_grammar :=                                             7
       $$dash2underscore(gfgrammar-AddressedApp-Language),
     asr_grammar :=                                            8
       $$dash2underscore(asrgrammar-AddressedApp-Language),
     devices := $all_devices/AddressedApp,                     9
     push( active_apps, AddressedApp ) ] ).                   10
```

The first condition (line 1) picks out the first move in the queue of non-integrated moves. Line 2 checks that the `app_per_move` variable is not empty (in case the interpreter's application indexing function is not in use), and line 3 picks out the application addressed by the current move to be integrated. Line 4 makes sure that the rule triggers only if the addressed application is not among the currently activated ones. The operations in lines 5-9 set the Resource Interface Variables to the appropriate values so that the addressed application becomes active[3]. Line 9 picks out the sub-record of `all_devices` containing the devices associated with the addressed application. Finally, line 10 pushes the addressed application onto the stack of activated applications.

User-initiated switching to an activated application If the user addresses an application which is among the currently activated ones, but not currently active (i.e. not on top of the stack of activated applications) we have a case of application reraising, initiated by the user. In this case, the addressed application is raised to the top of the stack. It is also necessary to raise all actions and issues associated with this application to the top of the issues and action stacks, so that the system will address these tasks when the currently requested task is finished, and as long as the addressed application remains active.

[3]The function `dash2underscore` takes a term and replaces all dashes ("-") with underscore symbols ("_"). This technicality provides a way of easily putting together terms that can be used as Prolog file and module names.

System-initiated switching to an activated application When a task within an application has been finished, the system will check whether there are other applications on the stack whose associated tasks that are not yet finished. If so, the application stack will be popped, removing the previously active application. The result is that a new application becomes topmost on the stack – this will now become the active application and the resource variables will be updated accordingly. The system will also issue a comment (a dialogue move of the form `icm:reraise:application(...)`)that a previously addressed application is now the active one (e.g. “Returning to the mp3 player”).

4.2.2 Grammars and language models for application accommodation

As explained in Section 4.1.4, implicit direct application switching requires that all applications can be updated off-line to learn about new applications, by adding a subset of new application grammar to all application grammars that will be connected simultaneously. We solve this by dividing each application grammar a *global* and a *local* part. The local part is only available when the application is active, whereas the global part is available in all connected applications. Section 5.3.2 explains how this is accomplished in GoDiS using GF[4]

In addition, it is necessary that the interpretation, generation and ASR modules are able to (1) load several alternative application grammars (or language models in the case of ASR) on start-up, and (2) alternate between these based on the currently active application. In an OAA-based system, this means that the GF and ASR modules need to offer solvables for setting the currently active grammar or language model. Given this, TrindiKit4 offers the possibility of setting up triggers that fire when the relevant Resource Interface Variables are modified, and that send out OAA solvables to the GF and ASR modules. These triggers are implemented as follows:

```
condition(val(asr_grammar,G)) => [ oaaSolve(setASRGrammar(G)) ]
condition(val(gf_grammar,G)) => [ oaaSolve(setGFGrammar(G)) ]
```

The Resource Interface Variables `asr_grammar` and `gf_grammar` are set by the update rules for application switching (see e.g. Section 4.2.1. The first trigger rule says that if the value of `asr_grammar` is modified and the result is that the condition holds that the new value is `G`, an OAA solvable is sent to via OAA facilitator to set the ASR grammar to `G`. This solvable will be handled by the ASR module agent.

4.2.3 Multi-device utterances

Multi-device utterances are utterances addressing several applications, i.e. utterances interpreted as addressing more than one application. The grammatically simplest case of this is sentence conjunction (e.g. “Stop the music and turn off the lights”). In case several connected applications share some vocabulary for requests (or questions), NP conjunctions may also occur (e.g. “Turn off the music and the lights”).

Handling such utterances requires the ASR and interpretation grammars to understand utterances in non-active connected devices; but as we have seen above, this is also required to allow for implicit direct

[4]The principle for which parts of an application grammar to include in the global grammar is chosen individually for each application. Essentially, it is a trade-off between device accessibility and ASR quality; if a large part of an application grammar is included in the global grammar, it will make the application easier to access from other applications, but it will also increase the size of the global grammar (including the ASR grammar) and thus the quality of the ASR output.

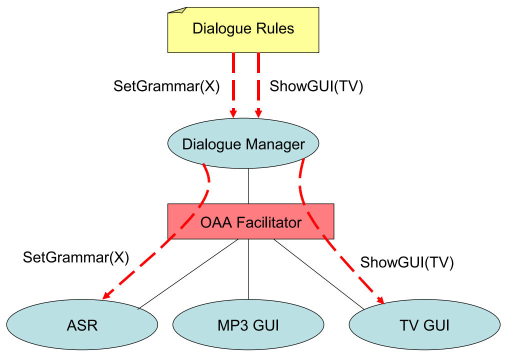

Figure 4.1: MIMUS Applications

application switching, so multi-device utterances pose no additional problems; the dialogue management and grammar solutions already proposed also covers multi-device utterances.

4.3 Multiple Applications in MIMUS

4.3.1 Taxonomy concepts applied to MIMUS

Application vs Devices

MIMUS allows for the ASR grammar and the GUI to be modifi ed at runtime, defi ning different contexts under which the user can control a set of elements. These contexts are the MIMUS equivalent for the previously defi ned "applications". As shown in fi gure 4.1, both the ASR grammar and GUI updates are controlled by the Dialogue Rules from the Dialogue Manager.

The set of elements controlled within a particular context is what the proposed taxonomy identifi es as "device". The MIMUS knowledge resources are defi ned as OWL ontologies, and "Device" is a particular Class of the Home Ontology. Clustering is achieved by means of SubClasses (i.e., Lamp is SubClassOf Device).

In MIMUS, application switching occurs in the same manner as any other multimodal event, that is, by speech, click or speech and click, as described in Deliverable 1.6. The user can switch between contexts (or rather applications) by voice and/or graphically.

MIMUS can work with both explicit and implicit direct switching. Indirect switching has been discarded because of the additional unnecessary step involved. In the following sections there are descriptions with examples of how application switching is achieved in MIMUS.

Tokens vs Types, Online vs Offline

The concepts of "token" and "type" could be assimilated, respectively, to the "Individual" and "Class" or "Property" concepts from the ontology. MIMUS has full "token-level (weak) plug and play" and partial

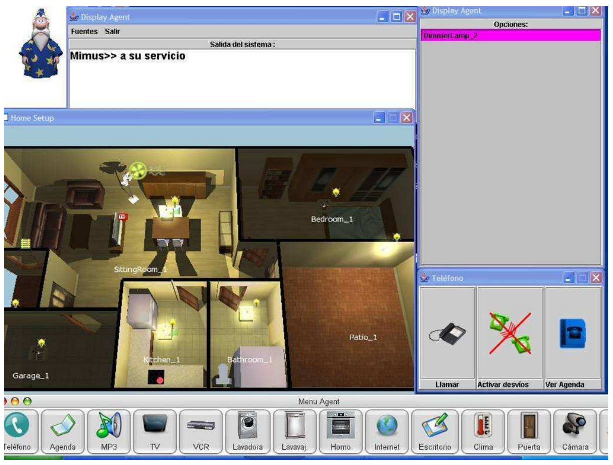
Mimus>> a su servicio

Figure 4.2: MIMUS Screenshot

"type-level (strong) plug-and-play".

The proposed terms "online" and "offline" are suitable for MIMUS, where offline and (partially) online reconfigurability is possible.

The way these concepts and classification is applied to MIMUS is described in section 5.2.

4.3.2 Multimodal application switching

MIMUS offers manyfold applications within the home scenario. As shown in figure 4.2, a set of icon–labelled buttons at the bottom of the screen grant graphical access to these applications to the user. These include the telephone, MP3 player and TV applications, among others.

The user can switch from one application to another by voice (*Show the TV interface*), graphically (Clicking on the TV Icon) or mixing modalities.

The use of different modalities can be done under MIMUS in three different ways:

Alternate inputs where the Dialogue Manager will ask for the piece of information missing:

```
U: (VOICE) Show me this.
S: Which interface do you want to load?.
U:(CLICK) TV Icon
```

Simultaneous combined inputs where the Dialogue Manager will fuse both inputs without asking for further information:

```
U: (VOICE) Show me this.
U: (CLICK) TV Icon
```

Simultaneous not related inputs where the Dialogue Manager will handle two independent tasks in parallel:

```
     U: (VOICE) Show me TV interface.
     U: (CLICK) Switch on (clicked on
     the submenu of a particular lamp icon)
```

4.3.3 Explicit direct switching

This is the case where the user explicitly asks for a particular interface. All of the utterances from the previous sections are examples of explicit direct switching. For instance:

```
U: Show me the MP3 interface.
S: OK.
```

This utterance from the user is parsed by the NLU module providing the following DTAC[5]:

$$\begin{bmatrix} \text{DMOVE} & \textit{specifyCommand} \\ \text{TYPE} & \textit{CommandShow} \\ \text{ARGS} & \begin{bmatrix} \text{INTERFACESPECIFIER} \end{bmatrix} \\ \text{INTERFACESPECIFIER} & \begin{bmatrix} \text{DMOVE} & \textit{specifyParameter} \\ \text{TYPE} & \textit{InterfaceSpecifier} \\ \text{CONT} & \textit{MP3} \end{bmatrix} \end{bmatrix}$$

Delfos receives this DTAC and triggers the corresponding rule:

```
(RuleID:        SHOW;
PriorityLevel:  15; TriggeringCondition:
        (DMOVE:specifyCommand,TYPE:CommandShow);

DeclareExpectations: {
    InterfaceSpecifier <= (DMOVE:specifyParameter,TYPE:InterfaceSpecifier);
    }

ActionsExpectations: {
    [InterfaceSpecifier] => {
    NLG(WhichDevice,@is-SHOW); /*Which interface do you want to load?*/
        }
    }
PostActions: {
    ExecuteAction(ShowGUI,@is-SHOW.InterfaceSpecifier.CONT);
    SetGrammar(@is-SHOW.InterfaceSpecifier.CONT);
    ActivateRule(FUNCTION);
    }
)
```

[5]For a detailed explanation on the DTAC protocol please refer to the SIRIDUS project, Deliverable 3.2.

It is within the PostActions of this rule where the application is switched, loading the accurate GUI (ShowGUI) and setting the Grammar (SetGrammar).

4.3.4 Implicit direct switching

The user can just say *Play a song* while being in another context (let's say the TV one). This utterance will trigger the **PLAY** dialogue rule:

```
(RuleID:        PLAY;
PriorityLevel:  15;
TriggeringCondition:
        (DMOVE:specifyCommand,TYPE:CommandPlay);
PreActions: {
    ExecuteAction(ShowGUI,"MP3");
    SetGrammar(".MP3");
    }

DeclareExpectations: {
    SongSpecifier <= (DMOVE:specifyParameter,TYPE:SongSpecifier);
    }

ActionsExpectations: {
    [SongSpecifier] => {
    NLG(WhichSong,@is-PLAY); /*Which song do you want to play?*/
        }
    }
PostActions: {
    ExecuteAction(Play,@is-PLAY.SongSpecifier.CONT);
    }
)
```

When this rule is triggered, the PreActions determine that the new MP3 application (grammar and GUI) has to be activated.

When a strategy allowing implicit direct switching is active, the main issue is to tune the different grammars allowing a partial set of utterances from different domains (i.e., letting the user say *Play a song* within the TV application). This tuning has to be done on a case by case basis, ensuring a trade–off between expressivity (the more utterances are available for the user in any context, the better) and ASR accuracy (the more restricted the grammar, the better WER from the ASR).

In the USE scenario for the MP3 application, we have chosen to keep the grammar as–is, only taking off from the lexicon the elements (songs, authors, etc.) from the MP3 database. In this case, when the user is within the TV application, he could say *Play a song* or *Load a new list*, and the system will understand the sentence, change the application, load the appropriate database lexicon, and engage in a specifi c subdialogue to accomplish the task.

In this scenario, USE currently has a lexicon composed by 1000 words. Consider a relatively small MP3 database with 1000 elements. Following the strategy previously described, we would have a 50% lexicon reduction, which will surely increase the ASR accuracy.

The cases studied within the home scenario let us believe that the approach of keeping a common set of grammar rules and reducing the lexicon size per application is suitable for any context employing implicit direct switching. Nevertheless, further investigations for different scenarios are needed for being able to generalize such a statement.

However, this is rather risky strategy when dealing with naïve users. Consider the case when the user says *Play Thriller by Michael Jackson* within the TV application. In this case, the ASR won't understand correctly because *Thriller* and *Michael Jackson* are not elements of the active ASR grammar. In this case, the expected behavior of the Dialogue Manager is to engage in a clarifi cation subdialogue for the piece of information missing. But the issue is that the ASR could give anything to fi ll the part of the sentence he could not understand, producing eventually 'valid' (from a grammar point of view) but incorrect utterances that the Dialogue Manager will work with. Our assumption is that this part of the sentence will have in most of the cases very low scores, allowing the Dialogue Manager to deduce that a clarifi cation subdialogue is necessary.

To sum up, again, there is a trade-off between completeness and robustness that only usability experiments and system tuning will help balance in the near future.

4.4 Applications in the Linguamatics Interaction Manager

Although the menu structure may have arbitrary groupings of devices or services, much of the clustering of devices and services is also done bottom-up. For example, if a new television is added to the system it appears under the "television" grouping. If the users says which room it is to be located in, it also will automatically appear under that room's grouping.

4.5 Multiple Applications in the Linguamatics Interaction Manager

The approach to applications in the Linguamatics system is rather different from GoDis and Mimus. Once an application has been plugged into the system it does not exist separately, except as a node in the hierarchy. For example, "cinema booking" might be added under "entertainment" which might itself by under "leisure". However, once added to the system, "cinema booking" is no more (or less) an application than "entertainment" as a whole or "leisure". Similarly, home control is not pre-defi ned as an application, but clustering of devices happens automatically according to the properties of the devices. For example, if a new television is added to the system it appears under the "television" grouping. If the users says which room it is to be located in, it also will automatically appear under that room's grouping. Although, as mentioned in 2.3.4, it is possible to impose more top-down clustering via a menu structure, the aim isfor this to be open to users of the system as much as to system designers.

Instead of a notion of "application", the closest notion in the Linguamatics system is that of "context". As a user traverses down "is-a" or "in-a" links the context changes and becomes more specifi c. For example, at the level of **kitchen** a mention of **light** will be assumed to be a mention of *kitchen-light* rather than lights in general. Similarly, if the user switches to a particular device such as the **radio**, an utterance of "turn the volume down" refers to the **radio** in the kitchen, not any other device. Although application switching is not a real notion in the Linguamatics system, there is a corresponding notion of context switching. Ideally the notion of context should affect the interpretation of a user's utterance (for example, by helping to pick the most plausible speech recognition hypothesis), but not fully determine it. Currently in the Linguamatics system, the context is a hard constraint, and utterances are always interpreted as narrower than the existing context. This means that the conversation can only get narrower and narrower until some action is taken, or the user explicitly goes back up the **is-a** or **in-a** links using fixed remarks such as "back up to . . . " or "back up to the top".

It is possible to perform commands over multiple devices or services, provided the user is in a context which includes both. For example, if in the **rooms** context it is possible to turn on the kitchen light and the lounge television. In principle, this should mean that at the top node you could say anything. However there are currently restrictions on the speech recognition grammars so that at the top node the whole grammar below is not accessible, but only the main nodes in the is-a hierarchy. Thus you can immediately skip to a lower node e.g. "cinema-booking", but cannot actually book the film. Ideally the whole grammar would be accessible, and the position in the context would act as a preference on hypotheses.

Chapter 5

Reconfigurability in the Systems

In this chapter we describe how reconfigurability is achieved in each of the systems, including the extent to which this can be done on-line for different components of the system.

5.1 Linguamatics Interaction Manager

Reconfiguring the Linguamatics Interaction Manager is primarily achieved by changing the ontology. This can be done both off-line or on-line by sending messages to the interaction manager. There are also off-line compile-time configuration files which specify how communication is achieved between different modules such as the recognizer, synthesizer and graphical output module, provide global defaults for the grammar (including escape words such as “Help”), and a configuration file for output (including XML tags if appropriate).

5.1.1 Dynamic Reconfiguration

Devices and services can be added or deleted from the ontology at run-time. When a new device is added, the information about the device e.g. the kind of device, its location etc. is sent in messages to the interaction manager, which then updates the main ontology.

Currently, the representation language used is a proprietary one which specifies nodes and their parents (rather than nodes and their children). Updating of the ontology with new nodes is therefore a monotonic operation, and can be achieved by just adding extra fragments of the ontology. An alternative representation which would be equally convenient would be OWL-RDF, which splits up information about the node and its relationships with other nodes.

For example, to add a new plasma television to the system results in the addition of the following information to the ontology:

> *tv-1* **is-a tv**
> *tv-1* **has-property** *plasma*

The device is located by asking the user for the room containing the new *tv*. This adds the information:

> *tv-1* **in** *bedroom-1*

This would be classifi ed as weak on-line plug-and-play according to 4, since this assumes that **tv** is a known concept.

To achieve strong on-line plug-and-play, information about the type of the device has to be sent along with information about the device itself. For example, if **tv** were not a known type, the following information would be sent:

> **tv is-a onoff-device** *tv* **is-synonym-of tv** *television* **is-synonym-of tv** *telly* **is-synonym-of tv tv has-state channel tv has-state soundonoff tv has-state volume**

This description still assumes knowledge about **onoff-device**, **channel** etc. To be completely confi dent that a new device can be safely added, all concepts referred to should include information about synonyms and their path (according to **is-a**) to the root node.

5.1.2 Current Limitations

Currently, images for devices are held in a confi guration fi le rather than in the ontology. This means that the graphical interface supports strong off-line plug-and-play, but only weak on-line plug-and-play.

A more major limitation concerns generation which is template-based. Although new templates can be sent to the interaction manager on-line, this approach is not well-suited to providing dynamically constructed answers which refer to multiple devices or services.

Finally, language modelling in the system is only partially reconfi gurable. The basic grammars are created from the ontology and are reconfi gurable on-line. However we also use a class-based statistical language model for home control, which is only partially reconfi gurable. If new devices are added to the ontology on or off-line, the system will then populate the classes appropriately in the language model. However, if a device has new commands, with different numbers of parameters from an exising command, the language model will not be automatically adapted. In future, we intend to adopt the approach which has been successfully explored in TALK Deliverable D1.3 [WJR$^+$06] of generating SLMs from grammars.

5.2 MIMUS

MIMUS approach to reconfi gurability is based on the idea of a central Ontology (eventually composed of several subontologies) managed by a Knowledge Manager (KM) agent. The confi guration changes are centralized through this Ontology.

In MIMUS there are three agents making use of this Ontology: the Home Setup (HS), the Dialogue Manager (DM) and the Device Manager (DevM). The HS loads the elements to be presented once at the begging of the execution (offline plug and play). The DM loads the grammar and lexicon also at the beginning of the execution (offline plug and play), but also queries the Ontology for every reference resolution call, allowing certain degree of online plug and play. The DevM receives at execution time petitions from the DM to update the device's state (e.g., from "on" to "off"). This agent sends the physical command to the real devices, updates the graphical layout of the house through the Home Setup and also the ontology which is dynamically reconfi gured by the KM. A schema of the inter-agents communication is shown in fi gure 5.1.

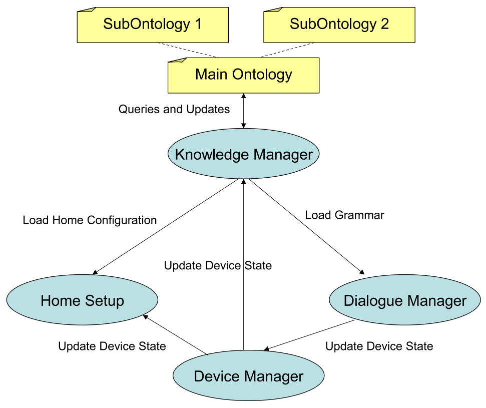

Figure 5.1: MIMUS Reconfigurability

5.2.1 Weak Plug & Play

This type of reconfi gurability in MIMUS corresponds to changes where no additional devices, services or functionalities are included. It may consist of:

- Changes in the current house set up such as moving a device to a different room
- Changing the device descriptors, i.e., the characteristic/s that may help distinguish each of them from other devices of the same type.
- Adding new devices, as long as they conform to the already existing device list. That is, the new device belongs to an already defi ned type and their descriptors already exist.
- Changes in the confi guration of certain services.

All these types of reconfi guration can be done at run time and imply changes in the ontology where all this information is contained. The ontology could either be accessed through a graphical interface (Prot´eg´e), or directly through an editor.

Since the devices and services here considered already exist and only new individuals are added to the ontology, there is no need to modify the grammars, lexicons or dialogue manager. The modifi cations therefore only affect the ontology and the Graphical Interface Manager (GIM), that is, the agent responsible for the graphical lay–out [1].

Ontology	Grammars	Lexicons	Dialogue Manager	GIM
YES				YES

[1]The Graphical Interface Manager in MIMUS is actually the Home Setup agent. However we leave the term GIM for generalization purposes.

Graphical Interface Update

In order to enable the system to update the graphical interface, the ontology has been extended to include graphical information. To be more precise, each device type has (in addition to all the information already included) an associated icon, which would in turn be the one to appear on the screen if a new such device were added.

This is necessary but not sufficient to update the graphical interface correctly. For the GIM to be able to include the new device icon in the appropriate location on the screen, the ontology also contains information about the graphical coordinates that correspond to each room or location in the house. In other words, each room in the house would have an associated set of coordinates that would delimit the area of the screen corresponding to its graphical representation. The following triplets illustrate these new relations.

$$\begin{tikzcd}
\boxed{\text{Device}} \arrow[r, "hasIcon"] & \boxed{\text{Icon}} \\
\boxed{\text{Room}} \arrow[r, "hasCoord"] & \boxed{\text{Coordinates}}
\end{tikzcd}$$

In consequence, should a new device of type Device, located in room Room be added to the ontology, a new icon of type Icon will appear within the coordinates of the room Room on the screen.

5.2.2 More complex reconfiguration cases

There are other possibilities to reconfigure the system that would imply more complex strategies and modifications at different levels:

1. Adding an existing functionality to an existing device
2. Adding a new functionality to an existing device
3. Adding a new device with new functionality

Adding an existing functionality to an existing device

In the first case, we are considering the possibility of extending the functionality of an existing device. However, the functionality considered in this case is already defined in the ontology as belonging to a different device. As an example of the case hereby described, let us consider two devices and two scenarios:

Scenario 1:

Device 1	VCR	Programmable
Device 2	DVD Player	Not Programmable

In this scenario, the VCR has a complex functionality that implies complex interaction with the user, specific vocabulary, grammar rules, dialogue rules and graphical displays. The DVD player, however, cannot be programmed.

Scenario 2:

Device 1	VCR	Programmable
Device 2	DVD Recorder	Programmable

In this scenario, the user has switched the old DVD player for a brand new DVD Recorder. New devices do not contain in themselves the necessary information to be automatically integrated in a Smart Home system; therefore at this point it would have to be the system that adapts to the new situation. The goal now is to manage the new DVD recorder.

Since the functionality to be associated to the DVD recorder is analogous or rather almost identical to that of the VCR, it seems natural to make use of the existing knowledge to associate the VCR functionality to the DVD recorder with the appropriate changes.

To do this easily, the ontology, lexicons, grammars and dialogue manager confi guration must necessarily be organized appropriately. That is, the grammar rules that determine what can reasonably be said must necessarily allow for commands such as *Record this movie on DVD*, or *Record this movie* and disambiguate with what device the movie should be recorded if more than one were available (like in scenario 2). In scenario 1, the system would reply to the user with something like *This device cannot record movies*, whereas in scenario 2, the system would already know that the new device has been enabled with this functionality. More generically, the system would be capable of understanding a range of commands, even if they do not apply to the current set up. In the following tables, the modules needing changes are marked.

Ontology	Grammars	Lexicons	Dialogue Manager	GIM
YES		YES		YES

Adding a new functionality to an existing device

In this case, the degree of complexity to reconfi gure the system may vary depending on the complexity of the functionality to be added. However, it is quite clear that all elements would have to be modifi ed:

Ontology	Grammars	Lexicons	Dialogue Manager	GIM
YES	YES	YES	YES	YES

Adding a new device with new functionality

As in the former case, as long as the devices themselves do not include the necessary information to interact with them at this level in a standard format, the system will have to be reconfi gured mostly manually and all elements involved would have to be modifi ed.

Ontology	Grammars	Lexicons	Dialogue Manager	GIM
YES	YES	YES	YES	YES

5.2.3 Applying the Taxonomy to MIMUS

Offline weak plug–and–play

This class of soft plug–and–play is fully available in MIMUS. Any new individual (token) can be added to the ontology, and the whole system will be automatically updated when rebooting MIMUS.

Offline strong plug–and–play

This kind of plug–and–play is only partially available. We can include new properties (for instance, hasHeight to identify the height of the Devices), and the grammar will be automatically updated, so that constructions from the user such as *What is the Height of the Kitchen Lamp?* will be understood. We can also include new Classes (for instance, **Fan** as subClassOf **Device**).
But in the general case, new types (Classes or Properties) would imply modifi cations on the dialogue rules and the Home Setup, which are not automatically updated when these modifi cations occur.

Online weak plug–and–play

This type of plug–and–play is again only partially available. Some individuals can be changed; the DM does not need to be aware of this change to process it correctly. For instance, say we have a blue lamp in the kitchen and the user asks about its color. As expected, the system would respond *blue*. If the ontology is now changed and the color is switched to red, the DM would answer appropriately *red* to the same question.
However, in order to obtain full online token–level plug–and–play, the Home Setup (HS) should refresh automatically changing the color of the lamp, which is not implemented yet.

Online strong plug–and–play

This kind of plug–and–play is not available in MIMUS.

5.3 Plug-and-play in GoDiS

This section presents the strategies and solutions implemented in GoDiS to enable plugging in new devices and applications.

5.3.1 Off-line strong plug-and-play for implicit direct application switching

In the terminology introduced in Section 4.1.1, the application plug-and-play strategy we have chosen to implement in GoDiS is *off-line strong plug-and-play for implicit direct application switching*.1

5.3.2 Adding applications to GoDiS

As we saw in Section 4.1.1, plugging in new application in the case of off-line application plug-and-play requires halting the system, and before restarting it taking the following steps:

- extending the application cluster is by adding one or several new applications to a list of applications to be run simultaneously
- getting all applications to (more or less automatically) exchange the appropriate information

These two steps are described in the following two sections.

Modifying the application cluster specification

In GoDiS, applications are connected by including the associated resources in the list of resources to be loaded on start-up (as specified by the predicate `selected_resources` in the application-cluster specification file (see the TrindiKit manual [SIR02]). This file also includes TIS updates to be carried out on start-up (specified by the predicate `reset_operations`); these should now include adding a field for the new application to the record of connected applications and their respective devices (`all_devices`; see Section 4.2.1).

Combining application grammars in GF

The application plug-and-play strategy chosen for GoDiS requires that applications exchange information enabling them to hear and understand when user wants to switch application. In terms of the grammar and ASR components, all applications can be updated offline to learn about a new application by adding a subset of the new application grammar to all application grammars that will be connected simultaneously. We do this by automatically extending GF grammars and compiling out new ASR language models. Once grammars and models for a certain combination of applications have been compiled, they can be re-used. Since we're doing off-line plug-and-play, it is not crucial that grammar combination and compilation is extremely fast (as it would be in the case of on-line plug-and-play).

The main idea is to split the grammars for user utterances into one ”global” and one ”local” grammar.[2] The global grammars contain utterances that every application should recognise, and the local grammars only those utterances that the active application should recognise.

Suppose we have the domains MP3 and Lights. Following the notation in the GF grammar library described in TALK deliverable D1.5 , the grammars for user utterances will be called `MP3User.gf` and `deLuxUser.gf`. The grammar library contains much more than these two grammars – there are grammars for system utterances, common grammars for all domains, and perhaps grammars describing the different entities in a domain – but the procedure in this section only have effect on these user utterance grammars.

The grammars `MP3User.gf` and `deLuxUser.gf` contains grammar rules for recognising user utterances. The first thing to do is to decide which of these should be global to any application and which should be local. The rules for global utterances are moved to the files `MP3Global.gf` and `deLuxGlobal.gf`, while the local rules remain in `MP3User.gf` and `deLuxUser.gf`.

[2]Note that the grammars for system utterances do not have to be shared, so this applies only to user utterance grammars

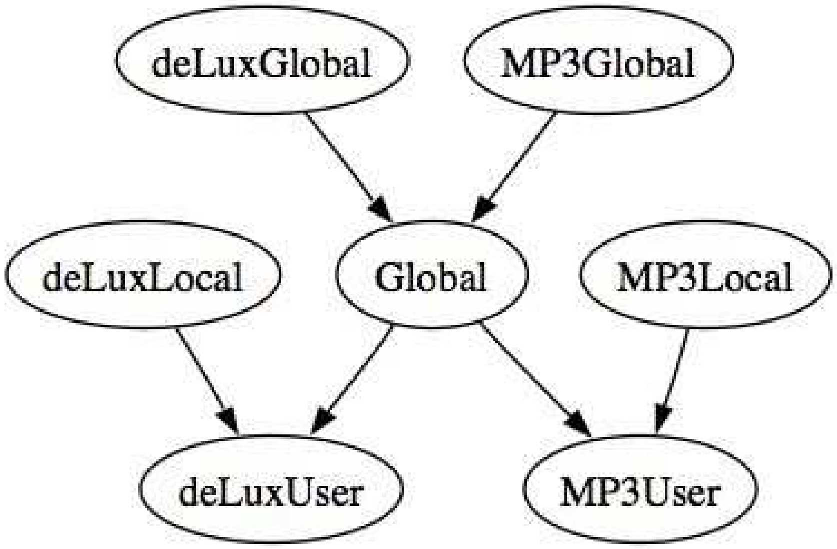

Figure 5.2: GF grammar structure for plug-and-play

The next thing is to create the new grammar Global.gf, which simply is the union of all global grammars. The file contains only one header line which imports the other grammars and does not add any new definitions:

```
abstract Global = MP3Global, deLuxGlobal ** {}
```

Now one single thing has to be added to the local grammars `MP3User.gf` and `deLuxUser.gf`. The grammars have to state that the Global grammar should be imported:

```
        abstract MP3User = ..., Global ** {
          ...
        }
```

(The first "..." are the grammars that the grammar already has imported, such as the common GoDiS grammar and other sub-grammars, the second "..." are the rules for local user utterances).

The final structure of the grammars is shown in Figure 5.2.

Now suppose that the Agenda domain is added, together with the grammars `AgendaSystem.gf`, `AgendaGlobal.gf` and `AgendaUser.gf`. There is only one grammar file that has to be changed, and that is the grammar `Global.gf` which now reads:

```
        abstract Global = MP3Global, deLuxGlobal, AgendaGlobal ** {}
```

Then we can recompile each of the user grammars into grammars for speech recognition and parsing, and calculate statistical language models.

The structure after adding the Agenda domain is shown in 5.3. Note that the only node that has changed is Global, since the incoming arrows are changed. All other nodes are the same as before.

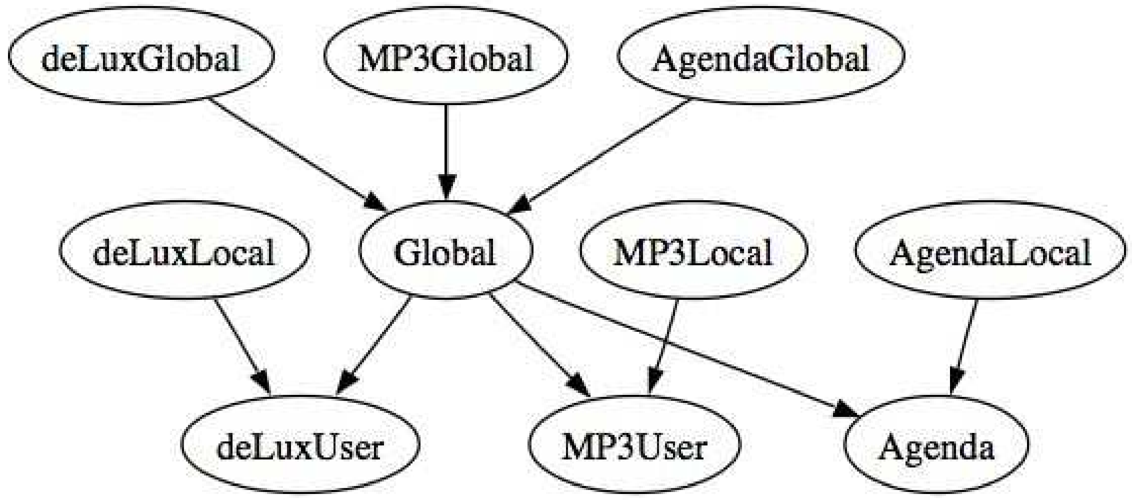

Figure 5.3: GF grammar structure for plug-and-play with additional application

5.3.3 Using OWL for application modifi cation and device plug-and-play

Section 2.4.2 shows how to specify a complete GoDiS dialogue application as an ontology in OWL. This description can than be automatically transformed into working GoDiS resources. Given such a specifi cation, GoDiS applications can be easily reconfi gured off-line using a standard OWL editor.

One way in which an application can be modifi ed is by plugging in a new device. Currently, there are no special facilities enabling plug-and-play of devices (within a given application) in GoDiS; essentially, the application needs to be reconfi gured (e.g. using an OWL editor) off-line. However, GoDiS' modularity minimises the effort required for plugging in a new device.

To add a new device of a new type to a GoDiS application off-line, one needs to add the associated domain knowledge to the application domain knowledge resource, and the GF grammar needs to be extended accordingly. All device resources are straightforwardly connected to the TIS by including them in the list of selected resources.

5.4 SAMMIE

In the SAMMIE dialogue system, our only focus was on reconfi gurability at the dialogue manager level — no effort has been made to explicitly support reconfi gurability in other system components (e.g., language resources). In this section, therefore, we only comment on reconfi gurability from the viewpoint of the dialogue manager.

The dialogue manager described in Section 3.2 does not yet support true dynamic reconfi guration, but rather offline reconfi guration. In order to reconfi gure the dialogue manager, the following information must be changed offline in the system:

Ontology New types in the CPS model (e.g., for *objectives* and *resources*) must be added to the ontology.

Grounded Recipes New Java classes representing *grounded-recipes* need to be added to the recipe package (detection can happen online).

The only bottleneck here to supporting dynamic reconfiguration is actually the ontology component, which does not allow changes at runtime. The Grounded Recipes are actually detected using Java reflection classes, and therefore could be added dynamically.

Although we have not described it here, our system relies upon an external database for knowledge of domain resources, and a new database interface would need to be added as well for reconfiguration. Although this is not a topic of our current research, we believe it would be possible to create a domain-independent component which could serve as an interface to any SQL database and return results as CPS resources. DFKI has created the MP3Shield component (see Deliverable 5.2) which does just that for the MP3 domain.

Chapter 6

Conclusion

In the Introduction to this report we outlined a series of challenges for multi-modal home information and control. Although no single system we have described meets all these challenges, we have shown how the information update approach embodied in different systems can meet the challenges, including the ability for easy reconfigurability, simultaneous conversations, and multi-modal multitasking.

In 2 we showed how ontological representations such as OWL are suitable for expressing a wide range of domain knowledge. In 3 we showed how domain-specific dialogue rules can be kept abstracted from generic dialogue management or separated into separate task processes. In 4 we discussed various kinds of plug-and-play and how multiple applications can be accessed simultaneously, or merged into a single structure. Finally in 5 we showed how it is possible to perform plug-and-play dynamically for devices and services.

Coping with the complexity of the home control domain has required the use of clean representations of dialogue context as provided by the information state update approach, and of domain knowledge as provided by ontologies. In showing the ability to plug-and-play new devices and services, we have shown the principles of how dialogue systems can become much more easily deployable and scalable in future.

Bibliography

[BA05] Nate Blaylock and James Allen. A collaborative problem-solving model of dialogue. In Laila Dybkjær and Wolfgang Minker, editors, *Proceedings of the 6th SIGdial Workshop on Discourse and Dialogue*, pages 200–211, Lisbon, September 2–3 2005.

[Bla05] Nathan J. Blaylock. *Towards Tractable Agent-based Dialogue*. PhD thesis, University of Rochester, Dept. of Computer Science, August 2005. Also available as University of Rochester Department of Computer Science Technical Report 880.

[Car90] Sandra Carberry. *Plan Recognition in Natural Language Dialogue*. ACL-MIT Press Series on Natural Language Processing. MIT Press, 1990.

[Cla96] Herbert H. Clark. *Using Language*. Cambridge University Press, 1996.

[CLAK05] K. Clarke, M.R. Lewin, D. Atkins, and R.S. Kalawsky. Testing a framework for multimodal control in the home environment. In *Proc. Perspectives in Pervasive Computing*, pages 87–95, IEE, London, 2005. DTI.

[LAC$^+$06] Peter Ljunglöf, Gabriel Amores, Robin Cooper, David Hjelm, Oliver Lemon, Pilar Manch´on, Guillermo P´erez, and Aarne Ranta. Multimodal grammar library. Deliverable 1.2b, TALK Project, February 2006.

[Lar02a] Staffan Larsson. *Issue-based Dialogue Management*. PhD thesis, Göteborg University, 2002.

[Lar02b] Staffan Larsson. *Issue-based Dialogue Management*. PhD thesis, Göteborg University, 2002.

[LBC$^+$05] Peter Ljunglöf, Björn Bringert, Robin Cooper, Ann-Charlotte Forslund, David Hjelm, Rebecca Jonsson, Staffan Larsson, and Aarne Ranta. The talk grammar library: an integration of gf with trindikit. Deliverable 1.21, TALK Project, July 2005.

[LCE01] Staffan Larsson, Robin Cooper, and Stina Ericsson. menu2dialog. In *Proc. of the 2nd IJCAI Workshop on Knowledge and Reasoning in Practical Dialogue Systems*, pages 41–45, Seattle, USA, 2001.

[MAB$^+$05] David Milward, Gabriel Amores, Tilman Becker, Nate Blaylock, Malte Gabsdil, Staffan Larsson, Oliver Lemon, Pilar Manch´on, Guillermo P´erez, and Jan Schehl. Integration of ontological knowledge with the ISU approach. Deliverable 2.1, TALK Project, September 2005.

[MB03] D. Milward and M. Beveridge. Ontology-based dialogue systems. In *Workshop on Knowledge and Reasoning in Practical Dialogue Systems*. IJCAI, 2003.

[Pro] Prot´eg´e. Prot´eg´e web site.

[Pro06a] TALK Project. Enhanced multimodal grammar library, July 2006. forthcoming.

[Pro06b] TALK Project. A unifi ed approach to multimodality and multilinguality in the in-home domain, December 2006. forthcoming.

[PS94] Carl Pollard and Ivan A. Sag. *Head-Driven Phrase Structure Grammar*. Studies in Contemporary Linguistics. University of Chicago Press, Chicago, 1994.

[Ran04] Aarne Ranta. Grammatical framework: A type-theoretical grammar formalism. *The Journal of Functional Programming*, vol. 14:2:pp. 145–189, 2004.

[Sir] Siridus. Siridus project web site.

[SIR02] SIRIDUS. Implemented siridus system architecture (enhanced). Project deliverable 6.4, SIRIDUS, 2002.

[TH92] David R. Traum and Elizabeth A. Hinkelman. Conversation acts in task-oriented spoken dialogue. *Computational Intelligence*, 8(3):575–599, 1992. Also available as University of Rochester Department of Computer Science Technical Report 425.

[Tra94] David R. Traum. A computational theory of grounding in natural language conversation. Technical Report 545, University of Rochester, Dept. of Comptuer Science, December 1994. PhD Thesis.

[W3C] W3C. Owl web ontology language overview.

[WJR+06] Karl Weilhammer, Rebecca Jonson, Aarne Ranta, Matt Stuttle, and Steve Young. Slm generation in the grammatical framework. Deliverable 1.3, TALK Project, February 2006.