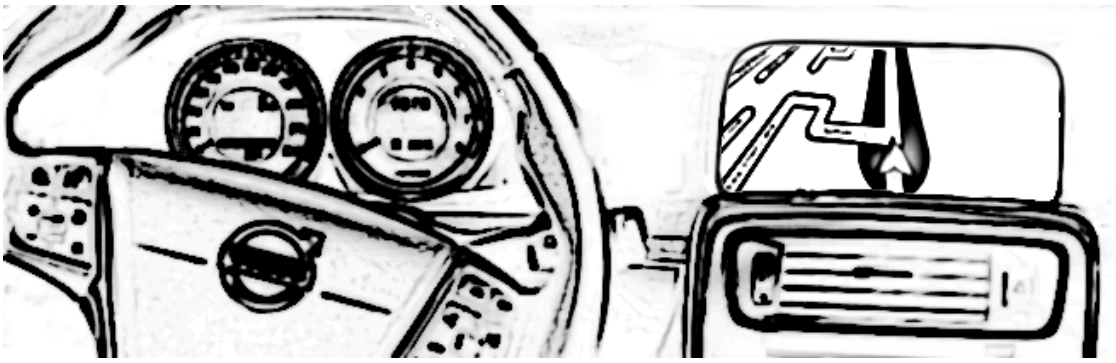
CHALMERS



# Constructing a Prototype Spoken Dialogue System for World Wide Named Places

Providing Spoken Dialogue Control of an In-Vehicle Infotainment System

*Master's Thesis in Intelligent Systems Design*


Robin Persson

Department of Computer Science & Engineering
Chalmers University of Technology
Göteborg, Sweden 2012

Constructing a Prototype Spoken Dialogue System for World Wide Named Places
Providing Spoken Dialogue Control of an In-Vehicle Infotainment System
Master's Thesis in Intelligent Systems Design

Robin Persson



Examiner: Peter Ljunglöf

Department of Computer Science & Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 1000

COVER:
The picture shows a car interior with an in-vehicle infotainment system. There is often a touch screen, located in the dashboard between the driver and passenger, allowing user interaction. There may also be a screen behind the wheel as well as a heads-up display shown on the windscreen, to display information to the driver. The picture is used on page 8.

Göteborg, Sweden 2012

**Abstract**

Named places, such as named restaurants and gas stations, are usually not supported in spoken dialogue systems used in vehicles. They usually support navigation in an in-vehicle infotainment system in the vehicle, but the user cannot mention named places.

This thesis attempts to construct a prototype dialogue system using an issue-based dialogue manager. The idea is to support named places across the world, in the context of driving a vehicle. The dialogue system is integrated with an in-vehicle infotainment system to enable spoken dialogue control of navigation and weather use cases.

The spoken dialogue system is grammar based and all the places used are loaded on startup. To enable world wide named places, new places are loaded dynamically at run-time, fetched from the cloud. This solution however introduces shortcomings, for example a long delay when waiting for the speech recognition grammar to compile. This problem also showed during user tests, being pointed out as one of the biggest problems with the prototype. Users also compared it to world-leading competition and pointed out the differences. The most important differences, on a conceptual level, can be solved by switching the grammar based speech recognizer for a freeform one. This solution would be interesting future work.

**Keywords:** spoken dialogue, infotainment, in-vehicle infotainment, speech, speech recognition, speech synthesis, grammar, navigation

# Acknowledgements

I wrote this thesis in the spring, summer and fall of 2012 at Pelagicore AB in Göteborg. I would like to thank them for inviting me into their office, assisting me and providing me with the tools I needed. In particular, I want to thank my Pelagicore supervisor Richard Röjfors and his occasional stand-in Jimmy Adler. I also want to thank the rest of the co-workers for providing valuable input during testing.

When writing my application for the Talkamatic Dialogue Manager I faced an undocumented work process. I had to tread where noone had treaded for a long time, discovering limitations and requirements along the way. The application, and this thesis along with it, would not have reached its final level of quality without the help of Talkamatic themselves. I extend my thanks, primarily to my Talkamatic supervisor Alexander Berman but also to his colleague, Fredrik Kronlid.

Finally, I would also like to thank my academic supervisor, Peter Ljunglöf, for guiding me with my early work as well as with this report.

Robin Persson, Göteborg 2012-11-23

# Contents

# 1

# Introduction

PELAGICORE[1] develops a resource framework (ResFW) for In-Vehicle Infotainment (IVI) systems, used in modern cars. They attempt to connect the varying resources normally available in an IVI system today, such as radio-, music- and movie playback, navigation, telephone integration, et cetera. Pelagicore also provides a human-machine interface (HMI), so that the user can interact with the IVI system.

Historically, IVI systems have been using physical means of interaction, such as buttons and knobs. However, recent days have provided them with touch screens and some kinds of speech control. Certain speech controlled systems principally lets the user navigate the IVI menu system by speaking menu labels out loud. Other IVI systems have a spoken dialogue (SD) component though, more oriented toward dialogue and allowing more natural utterances. However, to my knowledge, no speech controlled systems allow the user to mention places by name, for example specific restaurants or gas stations.

As pointed out by Chen et al. (2010), a spoken dialogue system is required to be highly natural, robust, seamless and highly intuitive and easy-to-use. It is hard to define if a spoken dialogue system is highly natural or highly intuitive, but one aspect to look at is place names. There are systems that allow places to be specified, it is however done either by narrowing down its category or specifying its address.

Another company, Talkamatic[2], develops a dialogue manager (TDM) for implementing flexible dialogue systems. It can be integrated with external systems, allowing development of dialogue control for them. It further models dialogue in a natural way, trying to imitate human-to-human dialogue in aspects relevant for SD systems.

[1] Pelagicore website: `http://www.pelagicore.com/` [Accessed June 12th, 2012]

[2] Talkamatic website: `http://talkamatic.se/` [Accessed June 12th, 2012]

## 1.1 Purpose

The purpose of this thesis is to allow users to control their car IVI system by speaking to it in natural dialogue, being able to name places.

A prototype demonstrator will be implemented, focusing on two use cases: navigation and weather. Both use cases are relevant to the context of driving a vehicle. The implementation will be introduced as a component of ResFW with TDM for dialogue management. Along with this, the goal is to also support world-wide named places, in context of the use cases. The use cases should both use named places so that the TDM naturalness can be demonstrated.

To provide an example, consider the use case of navigation. The user wants to navigate to a certain place, perhaps a specific restaurant in a city nearby. She requests navigation to the restaurant, mentioning its name while also mentioning the city name. TDM notices that the user is requesting navigation to some place in the nearby city. It tells the IVI to start navigation there. Next, the user asks what the weather will be like. TDM understands that she wants to know the weather for the destination she just mentioned. A weather forecast for the same place is shown in the IVI system.

## 1.2 Scope & Limitations

This thesis covers how TDM can be used to develop prototype dialogue control for an IVI system. Specifically, navigation and weather views of the HMI will be controlled in this way. There will be no spoken instructions from them though, they will look just like they do without spoken dialogue control.

Some user tests will be carried out on the prototype demonstrator to catch its major shortcomings.

While the system is an IVI system it will not be considered for a real automotive environment. In this thesis, a stable desktop environment will be used. For example, an Internet connection in a car is usually not constantly available, since the car can move across large areas, perhaps some with low Internet coverage. This is neglected in the thesis. Instead high speed Internet is considered constantly connected. Audible noise is usually common in cars, but in this thesis it will be considered non-existent, both from the environment and from passengers.

## 1.3 Outline

A prestudy of IVI and SD systems will be given in chapter 2. It should give proper understanding of the domains of the thesis. Chapter 3 will describe the existing components for IVI and SD systems that will be used in the prototype demonstrator. The demonstrator itself and the construction thereof will be described in two different versions in chapters 4 and 5 along with known shortcomings and discussions of them. An evaluation of the final demonstrator will then be described, including results and discussions in chapter 6. Finally, conclusions will be given in chapter 7.

# 2

# Technical Background

To get a better understanding of the domain of this thesis, some background of both spoken dialogue (SD) systems and in-vehicle infotainment (IVI) systems is needed. Both are described below but the part about SD systems focuses more on how such a system works, its components, et cetera. The part about in-vehicle infotainment systems focuses more on what such systems are capable of today, especially when integrated with a spoken-dialogue system of some sort.

## 2.1 Spoken Dialogue Systems

An SD system does not only interpret speech and speak back to the user. It can also keep track of the conversation, in contrast to more limited systems such as pure question-answer systems, where only the most recent question is considered and answered (McTear, 2002, p. 128).

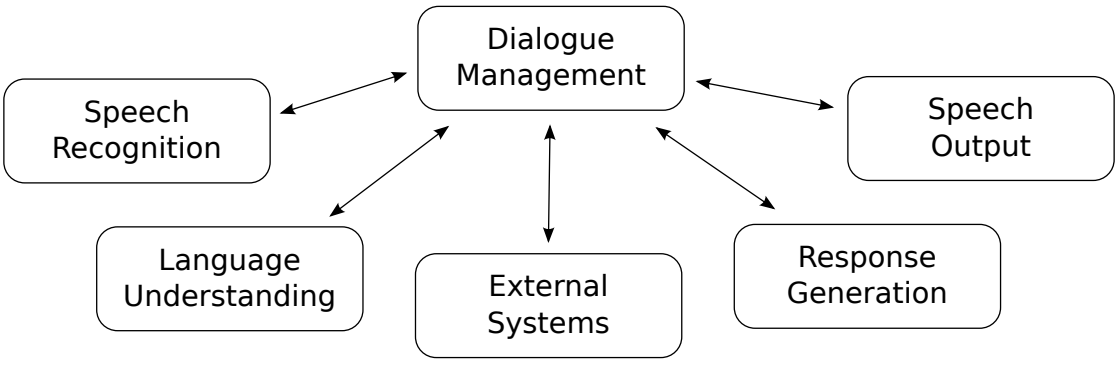



**Figure 2.1:** An SD system consists of six parts. Usually, dialogue management is central, controlling the other parts.

As described by McTear (2002, p. 113), a dialogue system mainly consists of six parts. Commonly, a dialogue manager (in the "Dialogue management" node) is central, controlling the other parts of the system, as can be seen in figure 2.1.

The automatic speech recognizer (ASR, in the "Speech Recognition" node of the figure) transcribes recorded user speech to text. The transcribed speech is semantically interpreted in the "Language understanding" node and external systems may be queried after this, if needed. "External systems" can be for example a multimedia player or a navigational system with GPS connectivity. A response is generated as text in the "Response Generation" node. The response can then be spoken, possibly by a speech synthesizer, in the "Speech Output" node.

I will describe speech recognition, speech synthesis and dialogue management below.

### 2.1.1 Speech Recognition

Speech recognition is important in SD systems. As stated by Kang et al. (2009), the speech recognition performance accounts for the biggest part of the overall SD system performance.

An ASR consists of models, trying to model speech from the real world. In addition, there is a speech engine which contains all the computational logic, making use of the speech models to transcribe speech into text. Two different models are needed for speech. Firstly, there is the low level acoustic model, modelling how words sound. Secondly, there is the language model, modelling how words build language.

#### 2.1.1.1 Acoustic Models

An acoustic model represents how words sound. It is commonly created by recording speech with text transcriptions, so that the speech and text can be matched. Statistical representations of how the words sound can then be computed. This means that the acoustic model is sensitive to the environment, such as noise and voice characteristics.

To achieve good speech recognition results, the acoustic model must be based on enough audio data, both regarding the amount of words and sounds spoken and the amount of speakers.

How speech can be modelled acoustically is described by for example Fant (2005, pp. 145-161).

#### 2.1.1.2 Language Models

Several approaches to language models for ASR systems exist. There are mainly two types, the grammar based model and the statistical model.

##### Grammar Based Language Models

Grammar based language models describe the domain language, trying to cover all potential utterances that a user may wish to speak. Utterances are not ranked in any way.

The ones that are finally recognized by the ASR are chosen entirely based on how well they match the acoustic model.

#### Statistical Language Models

Statistical language models (SLM) are usually based on probabilities of word sequences. Probabilities are calculated by counting occurances of words and word sequences in a corpus, a collection of text. For example, a new word can be predicted based on the previous word; or it can be predicted based on the two previous words. Using any number of words is viable but using one or two is the most common.

#### Grammar Based vs Statistical Language Models

A user may speak utterances that are not covered in a grammar based model. This usually means that grammar based models have a larger error than statistical language models, at least with naive users. However, if the user speaks an utterance supported by the grammar, the accuracy should be significantly higher for a grammar based model than an SLM since there are fewer potentially matching utterances to choose from.

Since SLMs can help the ASR recognize any utterance, not just those specified in a grammar, they are considered more robust than grammar based models. Of course, the SLM corpus must be large enough to include such an utterance and the domain needs to be considered when selecting the corpus. For example, a corpus based on news reader transcripts may not be the best for recognizing navigation requests in an automotive setting.

### 2.1.2 Speech Synthesis

A speech synthesizer, or text-to-speech (TTS) system, turns text into speech. Usually it will speak just by supplying it with plain text. This can be done by developing speech models, modelling how speech sounds based on transcribed recordings (Acapela, 2012a).

As further described by software synthesizer company Acapela (2012b), the most important qualities of a speech synthesizer is naturalness and intelligibility. Naturalness measures how natural the speech synthesizer sounds, how close to a real human being it is. Intelligibility on the other hand measures how well the user understands what the speech synthesizer has said. A good speech synthesizer must be both natural and intelligible.

## 2.2 Dialogue Management

A dialogue manager interprets the user's input in the context of the dialogue, keeping track of what has been said and deciding what to say next. The dialogue manager simply controls the behaviour of the dialogue system. There are several approaches to dialogue management. Some are well-tested, some are newer. I will describe two well-tested

approaches, one based on finite states and one based on frames. Among the recent ones, I will describe one based on issues.

### 2.2.1 Finite State Based Dialogue Management

In a finite state based dialogue manager, the dialogue moves between states. Certain answers need to be provided for the system to reach its goals, each answer bringing the dialogue to a new state towards the goal. A state reflects what information that have been provided and all possible answers that will move the dialogue to new states. The downside with this approach is that the state machine easily becomes complex and hard to handle when trying to model natural dialogue. The upside however, is that speech recognition results are rather accurate since potential answers are heavily restricted in individual states. (McTear, 2002, pp. 92-93)

An example of the finite state approach is the CSLU toolkit by Sutton and Cole (1997).

### 2.2.2 Frame Based Dialogue Management

In a frame based, or form-filling, dialogue manager; dialogue is modelled as a form, with a set of fields that need to be filled. The system requests the remaining fields until they are all filled. In light of this, the form filling model is better suited for mixed initiative dialogue, where both the user and the system can drive the dialogue. The user can answer more questions than one at a time, and the order of the answers is not important. (McTear, 2002, pp. 93-94)

An example of the frame based approach is VoiceXML (VoiceXML Forum, 2000), a markup language used to describe frame based dialogue.

### 2.2.3 Issue Based Dialogue Management

In an issue based dialogue manager, issues (or questions) arise in the dialogue and need to be resolved (answered). They may be both raised and answered by either the user or the system. Dialogue is flexible as well. The user may for example start the dialogue by answering an issue (rather than raising it) and, if there are several issues that accepts the answer, it will ask which one she means.

An example of an issue based dialogue manager is GoDiS, by Larsson et al. (2000). A spinoff of the GoDiS dialogue manager is used in the thesis and will be discussed in section 3.1.

## 2.3 Evaluation of Dialogue Systems

Several evaluation methodologies for SD systems have been suggested in literature. I will describe one method, PARADISE, by Walker et al. (1997). In the paper, Walker et al. mention some evalution measures that seem to be useful estimators in SD system

evaluation. Among them, task completion rate and concept accuracy, which will also be described below.

The final prototype demonstrator (chapter 5) will be evaluated briefly in chapter 6, using both task completion rate and concept accuracy. PARADISE itself, will not be used in the thesis.

### 2.3.1 PARADISE

User tests usually require much time and effort, associated with high costs. Because of this, methods to make user testing cheaper have been researched. One resulting method is PARADISE by Walker et al. (1997). They suggest extensive user tests initially, using an algorithm to correlate user satisfaction (which they consider the overall priority of a SD system) with a small set of the data gathered during testing. This way, user satisfaction can be estimated cheaply during later tests, as there are fewer parameters that needs to be tested.

### 2.3.2 Task Completion Rate

Task completion rate can be described as how efficiently a user completes a task. In this case how many utterances that are required to solve a certain task. The completion rate can be calculated individually for each task as well as for the system as a whole.

### 2.3.3 Concept Accuracy

To determine what concept accuracy is, the meaning of concept must first be established. Consider an utterance such as:

```
I want to navigate to New York.
```

There are two concepts in it. The first is to start a navigation. The second is the name of the destination, `New York`. Note that concepts are determined by the domain they are used in, as well as how they are used in the dialogue manager. In the thesis, concepts work as described here.

As described by Boros et al. (1996), concept accuracy (CA) refers to how big share of concepts that the system interprets correctly out of those given by the user.

## 2.4 In-Vehicle Infotainment Systems

This section covers In-Vehicle Infotainment (IVI) systems and their functionality, especially focusing on speech interaction. I have no insight in how an IVI system is normally designed, but I have studied some functionality. In particular, I have studied three existing IVI systems. Two of them were ordinary IVI systems (Volvo Sensus and Cadillac Cue), while one was a SD component (Ford Sync) that was part of an IVI system.

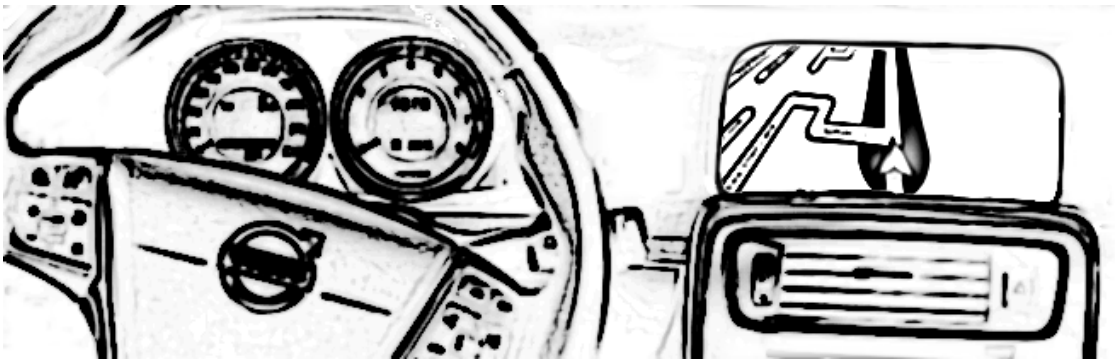


**Figure 2.2:** This is an example of what is usually associated with an in-vehicle infotainment system. There is normally a touch screen, located in the dashboard between the driver and passenger, allowing user interaction. There may also be a screen behind the wheel as well as a heads-up display shown on the windscreen, to display information to the driver.

### 2.4.1 Volvo Sensus

The Volvo Sensus[1] provides for example control of many Volvo safety systems, but also multimedia playback, bluetooth support and navigation. The navigation system is interesting. Volvo claims it has voice recognition[2] but there is no information of how it works. In a review though (Fung, 2011), it turns out that navigation destinations cannot be entered by voice at all, unless the user creates some kind of tags in advance, one for each destination.

### 2.4.2 Cadillac Cue

The Cadillac Cue[3] provides similar features, for example voice recognition for music, phone and navigation[4]. A review by Howard (2012) compares the Cue with the Ford Sync (see section 2.4.3) and determines that both of them are limited because their commands are processed locally, within the car, rather than on a server in the cloud. The car computer is not as powerful as a server solution in the cloud, which restricts both systems. The reviewer uses an example of a named place as something none of them can handle. This strongly suggests that both of them use static grammars rather than SLMs or dynamic grammars.

[1] American Volvo website, Sensus: `http://www.volvocars.com/us/sales-services/sales/soundandnavigation/pages/volvo-sensus.aspx` [Accessed Oct 10th, 2012]

[2] American Volvo website, navigation system: `http://www.volvocars.com/us/sales-services/sales/soundandnavigation/VolvoNavigationSystems/Pages/Built-In-Navigation-System.aspx` [Accessed Oct 10th, 2012]

[3] Cadillac Cue website: `http://www.cadillac.com/cadillac_cue.html` [Accessed Oct 10th, 2012]

[4] Cadillac Cue website, cue facts: `http://www.cadillac.com/cadillac_cue.html#nav_tablay_item_c1_1_1` [Accessed Oct 10th, 2012]

### 2.4.3 Ford Sync

Ford Sync[5] is not an IVI system by itself, rather a SD component. The Sync enables users to ask about anything from weather and navigation to music artists, gasoline prices and phone text messages[6].

I have never tested Sync myself, but judging from available information, it seems to use a static grammar. For instance, a review of the Ford Sync (Howard, 2012) determines that it is impossible to start a navigation to named places, using their names. There is a way to find specific named places, but the user needs to specify them correctly, without using their names. An utterance example that works with Sync is ”find an organic restaurant”, mentioned on the website[6].

[5]Ford Sync website: `http://www.ford.com/technology/sync/` [Accessed Oct 10th, 2012]

[6]Ford Sync website, showing available commands: `http://www.ford.com/technology/sync/sync-commands/` [Accessed Oct 10th, 2012]

# 3

# Review of Existing Components

All components that were used in the thesis but were not developed as part of it are described in this chapter. The purpose of the chapter is not to review components of the same sort, but merely to review the components used in the thesis prototype demonstrators. Specifically, the Talkamatic Dialogue Manager (TDM), Grammatical Framework (GF), D-Bus, the Pelagicore Resource Framework (ResFW) and the Pelagicore human-machine interface (HMI) are described herein. The final prototype, using these components, will be described in chapters 4 and 5.

## 3.1 Talkamatic Dialogue Manager

Talkamatic Dialogue Manager (TDM) is an issue based dialogue manager, as described in section 2.2.3. It is written in Python[1], being developed at Talkamatic[2], a spinoff company from the University of Gothenburg, where a TDM predecessor started off under the name of GoDiS (Larsson et al., 2000).

TDM has existing connections to ASR and TTS as seen in figure 3.1. The PTT adapter shown in the figure can be connected to an external system to enable a push-to-talk (PTT) button. Components of TDM that are relevant for the thesis and central in issue based dialogue management will be described in section 3.1.1.

To make dialogue flexible, TDM supports the concepts of accommodation (section 3.1.2) and grounding (section 3.1.3). Accommodation enables the user to suddenly raise a new issue in the dialogue. The system will handle the new issue before moving back to the previous one. Grounding, on the other hand, is a way of giving feedback to the user, to make sure the system and the user both know what has been said in the dialogue.

There are also TDM applications, which a dialogue system developer needs to create

[1] TDM works with Python v2.7.3. See the documentation: `http://docs.python.org/` [Accessed Aug 7th, 2012]

[2] Talkamatic website: `http://talkamatic.se/` [Accessed June 12th, 2012]

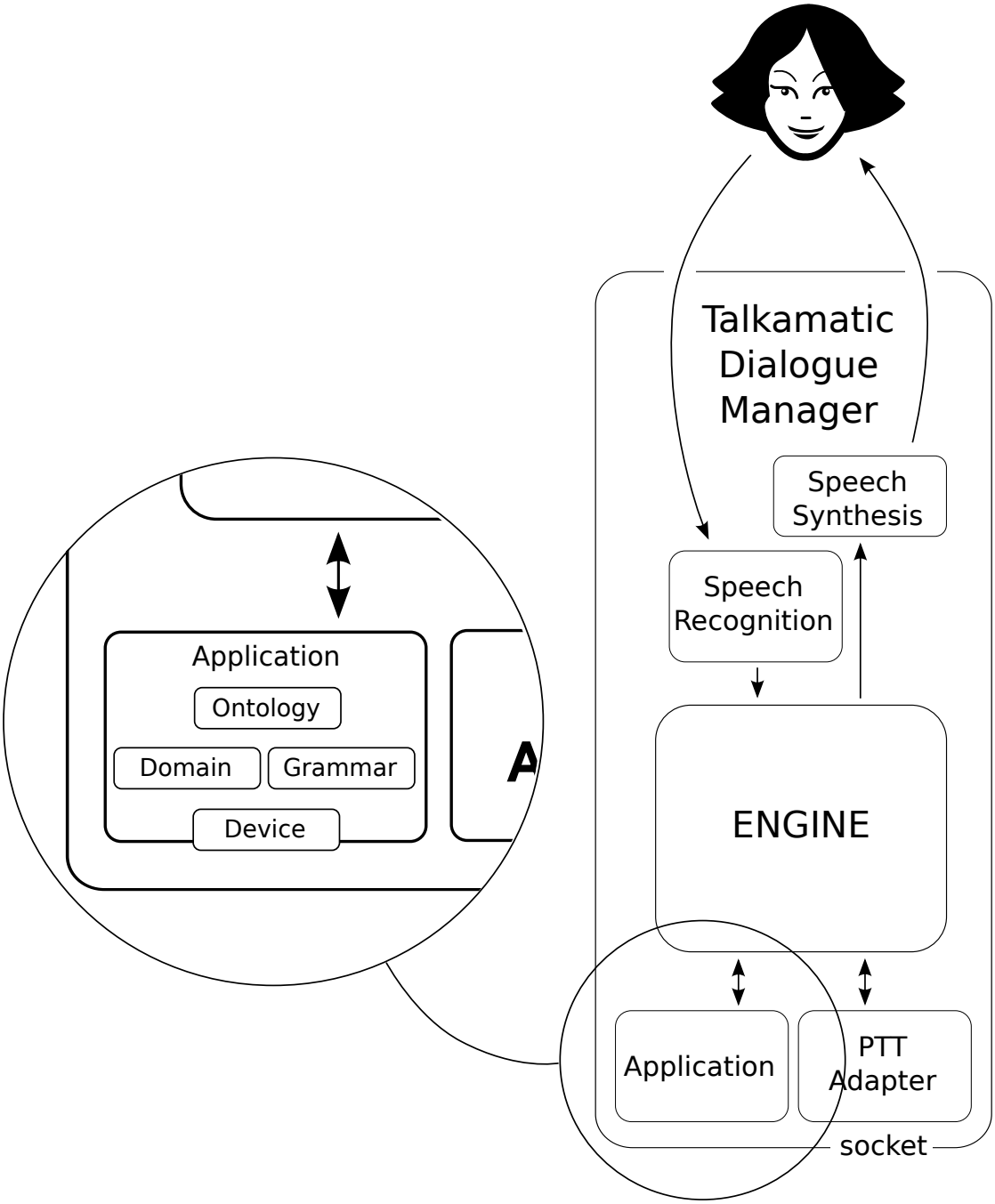



**Figure 3.1:** TDM wraps the ASR and TTS systems. The engine here means the functionality related dialogue management while the application and PTT adapter are shown to visualize the entry points for external communication.

in order to construct a dialogue system. In an application, he can specify actual language and what it means, as well as communicate with external systems. TDM applications will be described in section 3.1.4.

Larsson (2002) describes a variant of GoDiS, called IBiS, providing much of the same functionality. Both IBiS and GoDiS are written in Prolog (Larsson, 2002, sect. A.3), whereas TDM is written in Python. The thesis by Larsson is the best formal documentation there is for TDM, with the implication that it is not entirely accurate. TDM itself is documented through some basic tutorials on how to write applications as well as through its Python code, which is supposed to be self-explanatory. There is no more documentation because TDM is an ongoing development project yet to be released. When IBiS and TDM differ I will use my personal experience to describe TDM as well as I can. Where it however can be backed by the thesis of Larsson, so will be noted.

### 3.1.1 Components

There are many central components of TDM (Larsson, 2002, sect. 2.7.1), components that are also central for issue based dialogue managers in general. The components that are relevant for the thesis are dialogue moves that describe semantics, issues that correspond to questions, plans that describe which plan items that are needed to resolve an issue and then the plan items themselves. All are described shortly below. For information about the remaining components, please refer to the PhD thesis by Larsson (2002).

#### 3.1.1.1 Dialogue Moves

A dialogue move describes the semantics of an utterance. It corresponds to the concept mentioned in section 2.3.1. In the thesis, dialogue moves and concepts are exactly the same.

In TDM, there are several types of dialogue moves, for example the `request`, `ask` and `answer` moves. The `request` move indicates that the user initiates an action (Larsson, 2002, sect. 5.3.2). This will lead to TDM trying to execute a plan for that action. The `ask` move indicates that a question is raised and the `answer` move indicates an answer to such a question (Larsson, 2002, sect. 2.5).

Regardless of type, the contents of dialogue moves are represented as a simplified predicate logic. They consist of a predicate and a constant. The predicate describes the constant in context of the domain, while the constant contains the actual contents (Larsson, 2002, sect. 2.4.2). Constants of questions however, are unknown and need to be answered. In this case, the constant is replaced by a variable, which is represented by an `x`. There are three types of questions (Larsson, 2002, sect. 2.4.3).

- **y/n-questions** – Represented by a question mark in the semantics. For example

  ```
  ?destination(new_york)
  Do you want to navigate to New York
  ```

`destination` is the predicate, `New York` the constant.

- **wh-questions** – Represented by a question mark and a variable `x`. For example

  ```
  ?x.destination(x)
  Where do you want to navigate
  ```

  Again, `destination` is the predicate.

- **Alternative questions** – they are sets of y/n-questions. For example

  ```
  {?destination(new_york), ?destination(new_jersey)}
  Do you want to go to New York or New Jersey
  ```

How the different types of dialogue moves are commonly used in dialogue can be visualized by examples. Consider that the user wants to navigate to New York.

Firstly, the user asks the dialogue system to navigate to a location, interpreted as a `request`-move.

```
U> I want to navigate
request(action(startnav))
```

Secondly, if the dialogue system does not know the destination location, it will ask for it. It turns an `ask`-move into speech.

```
ask(?x.destination(x))
S> Where do you want to navigate?
```

Thirdly, the user responds with the destination, interpreted as an `answer`-move.

```
U> New York
answer(new_york)
```

A request as the one above can also be given in one go. If so, the utterance is interpreted as several dialogue moves.

```
U> I want to navigate to New York
request(action(startnav)), answer(new_york)
```

#### 3.1.1.2 Issues and Plans

Issues are raised during the dialogue, by the system or the user. An issue can be raised by requesting an action or providing an answer that is part of the plan of the issue.

Once an issue is raised, it needs to be resolved. This is done by completing its plan (Larsson, 2002, sect. 2.5). Two examples of issues with their corresponding plans are shown in figures 3.2 and 3.3. They will be used in the examples below.

TDM needs to infer which issue that the user is talking about when she provides a dialogue move. If the user starts the dialogue with an `answer`-move for example, TDM might not know which issue that is being answered.

```
ISSUE : startnav
PLAN:
  findout(?x.nav_destination(x))
  dev_perform(?x.startnav(x))
POSTPLAN:
  findout(?x.domain(x))
  change_domain(?x.domain(x))
```

**Figure 3.2:** This is an example of a plan to start navigation to a location. The user destination is required, specified as a findout plan item. When the destination is known, the dev_perform plan item makes sure an actual navigation is started for the user to see. The change_domain plan item loads the TDM application given by the preceding findout.

```
ISSUE : weather
PLAN:
  findout(?x.weather_location(x))
  dev_perform(weather))
POSTPLAN:
  findout(?x.domain(x))
  change_domain(domain)
```

**Figure 3.3:** This is an example of a plan to check the weather at a location. The location is required, specified as a findout plan item. When the location is known, the dev_perform plan item makes sure a weather forecast is fetched and shown to the user. The change_domain plan item loads the TDM application given by the preceding findout.

```
U> New York
answer(new_york)
```

TDM will go through all issues, looking for those that need the given answer. If there are more than one matching issue, TDM will ask the user which issue she actually means.

```
ask({?action(startnav), ?action(weather)})
S> Do you want to start navigation or show the weather?
```

Plans contain sets of plan item. Most common are the `findout` and `dev_perform` items, but there is also the `change_domain` item that will be relevant for the thesis. There is also a postplan that can be declared with a plan. The postplan is always executed after the plan has been completed.

All items of the plan, as well as of the postplan, need to be executed before the issue behind the plan itself can be considered resolved.

### findout plan items

If a question needs to be answered, this is specified with the `findout` plan item (Larsson, 2002, sect. 2.6.2). Questions can be answered by either the user or other systems.

1. Check if the answer is already known.
2. Ask all available device resources of the application (see section 3.1.4) if any of them knows the answer to the question.
3. As the last resort, ask the user a direct question to find the answer.

#### dev_perform plan items

Another type of plan item is the `dev_perform`. It executes a device action, specified in the device resource of the application (see section 3.1.4). A device action can communicate with an external system, for example when the user wants to start a navigation somewhere. The actual navigation is carried out by a navigation system, likely showing a map and route in a graphical user interface. For example, when the `dev_perform` item in figure 3.2 is executed, TDM will tell the navigation system to start a navigation.

#### change_domain plan items

The final type of plan item is the `change_domain`. It loads a new application into TDM, removing the currently loaded one. This means that everything previously learned for the domain is forgotten.

### 3.1.2 Accommodation

Accommodation is the concept of addressing unexpected dialogue moves by the user. It occurrs for example when the user provides an answer to an issue that has not yet been raised (Larsson, 2002, sect. 4.9). TDM will try to find which issue the user means, if there are more than one that needs the answer. If there is only one matching issue, TDM needs to make sure that the user is really talking about this issue. This is one example that distinguishes TDM from frame based dialogue managers (section 2.2.2). They require a frame (or issue, when comparing to TDM) to be resolved before raising a new one. An example of accommodation in TDM is the example in section 3.1.1.2. Another example is this:

```
U> New York
S> Do you want to navigate or show the weather?
U> Navigate
S> Started navigation to New York
```

### 3.1.3 Grounding

Grounding is the concept of making sure that all dialogue participants share the same knowledge of what has been said, establishing it as part of common ground. Grounding is done through spoken feedback to the user, on several levels. There is for example pessimistic as well as optimistic feedback (Larsson, 2002, sect. 3.6.1). Pessisimistic feedback explicitly asks the user if what it heard was correct. For example:

```
U> I want to navigate to New York
S> You want to navigate to New York. Is that correct?
```

Optimistic feedback on the other hand, implicitly verifies that it heard correctly. The user does not need to reply, unless TDM misheard her. For example:

```
U> I want to navigate to New York
S> Started navigation to New York
```

The type of feedback TDM gives depends on how certain it is of what it heard. Usually, an ASR gives a score of recognition accuracy along with its results. The score is used to determine the feedback type to use for TDM. If TDM is really certain of what it heard, it may be enough with a simple `Okay`. If less sure, it may give optimistic feedback while pessimistic feedback is needed if unsure. If the score is really low or there are no results at all from the ASR, TDM will tell the user that it did not hear the utterance.

### 3.1.4 Applications

A TDM application is not part of TDM, but constructed by a developer for a specific project and domain. I have for example built a TDM application as part of the thesis. Applications are written in Python, just like TDM.

To use an application with TDM, it must first be compiled by the TDM compiler. The compiler compiles the application on two levels, one for TDM and one for the ASR.

The application itself contains four resources, shown in figure 3.1. The ontology resource is the core. It is used to declare predicates, constants and actions so they can be used in plans and in TDM. The grammar resource maps semantics to a specific language while the domain resource specifies plans. The device resource handles communication with external systems. For further details, see the corresponding sections below.

#### 3.1.4.1 Ontology Resource

The ontology resource is used to declare predicates, constants and device actions for use in TDM, for example in the grammar, domain and device resources. If not declared in the ontology resource, they cannot be used in TDM.

#### 3.1.4.2 Grammar Resource

The grammar resource specifies how predicates, constants and device actions from the ontology resource are expected in user speech. The grammar resource also specifies how the system will speak back to the user. For example, how questions are formulated is specified here.

One could say that the grammar provides a mapping between speech and dialogue moves (Larsson, 2002, sect. 2.5).

#### 3.1.4.3 Domain Resource

The domain resource in TDM is used to declare all the plans needed in the application.

#### 3.1.4.4 Device Resource

The device resource is used to declare device actions and provide answers to questions. If a `dev_perform` plan item is executed, a corresponding device action in the device resource will be executed. Similarily, if a `findout` plan item is executed (a question needs to be answered) the device resource can provide an answer, it will do so before the question is asked to the user. Thanks to being written in Python, the device resource can interact with external systems as far as Python allows.

## 3.2 D-bus

D-bus[3] is a message bus system for interprocess communication. It allows applications to communicate with other applications, running in separate processes. It also allows applications to send broadcast messages, accessible to any interested listener.

There are bindings to many frameworks and languages, for example Qt, GLib, Java, C#, Python, et cetera. It is widely spread in the UNIX world and may soon have support for Windows as well.

## 3.3 Pelagicore Resource Framework

The Pelagicore Resource Framework (ResFW), developed by Gothenburg based Pelagicore[4], is a framework to connect resources in an IVI system. Approaching ResFW top-down, you can see in figure 3.4 that the HMI is separated from ResFW, communicating with it over D-bus. The idea behind ResFW itself is that any HMI can connect to arbitrary automotive resources through a common interface in ResFW. Resource implementations can be replaced or updated without the user noticing. For example, ResFW currently supports two different bluetooth stacks, Bluez and BlueGo. They can both be used although just one at a time, without the user knowing which one. As another example, ResFW currently supports music playback from both Spotify and Pandora. They can be joined together seamlessly, providing music playback without the user knowing from which source.

ResFW is written in Python[5] and communication with the resources is usually done over D-bus. The infrastructure in ResFW allows it to keep track of resources as they become available or unavailable. Even if a resource is available, it may not be running. But as soon as the HMI requests it, it will be started by ResFW.

Declaring a resource means declaring it in two parts. One is the interface toward the HMI, specific for a certain resource type. This can for example be media, providing an abstract interface regardless of the actual media source beneath. The second part is the integration of the actual resources. In the media source example, both Spotify

[3]See the wiki at: `http://dbus.freedesktop.org/` [Accessed June 12th, 2012]

[4]Pelagicore website: `http://www.pelagicore.com/` [Accessed June 12th, 2012]

[5]ResFW works with Python v2.7.3. See the documentation at: `http://docs.python.org/` [Accessed Aug 7th, 2012]

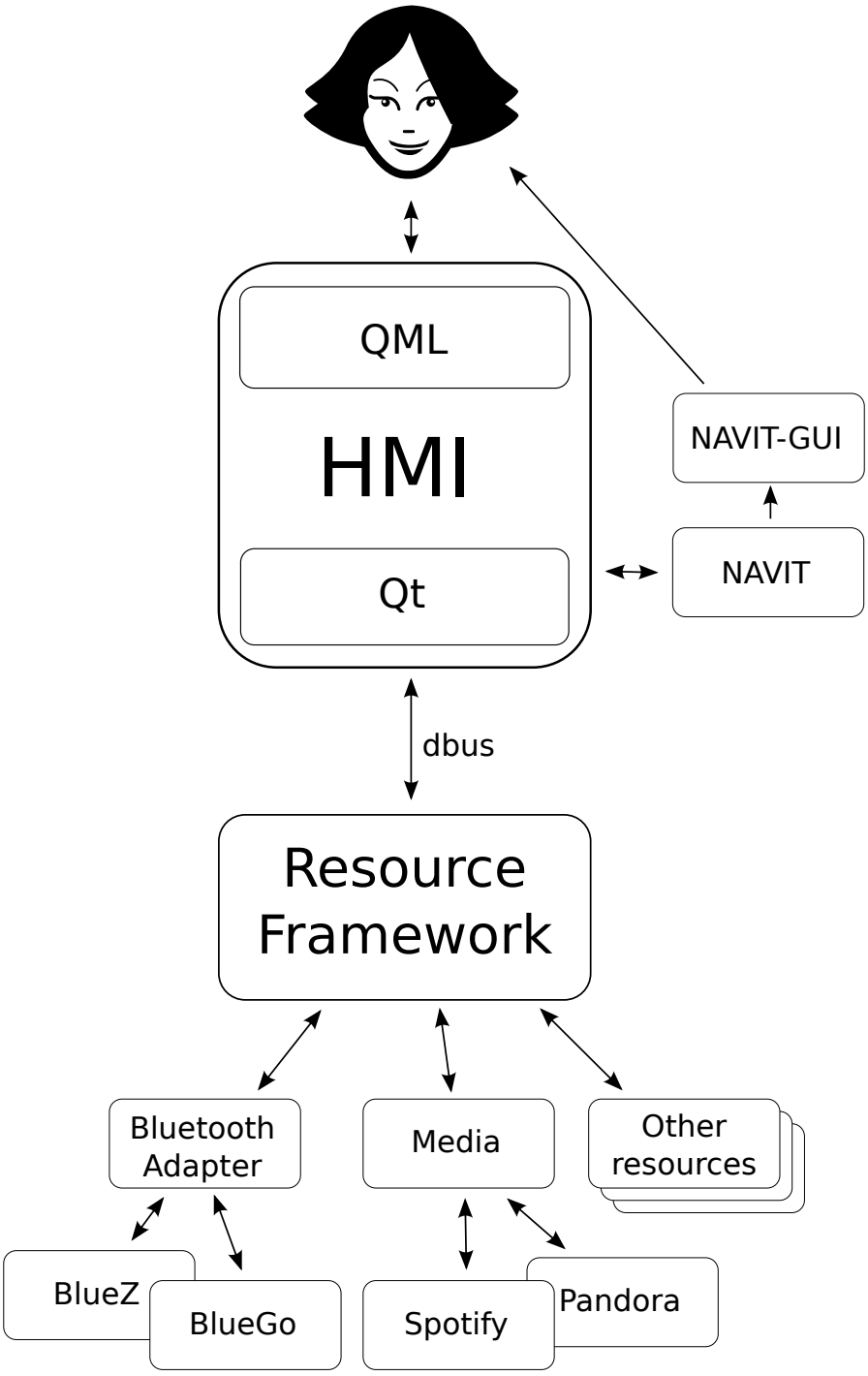



**Figure 3.4:** ResFW connects all kinds of resources, especially several of the same kind, providing a common interface toward the HMI. Open-source navigation system Navit was used as a navigation resource in ResFW but was not implemented as a real resource. Instead, it could be accessed directly by anyone across D-bus.

and Pandora are integrated with ResFW. Adding a new media source only requires integration with ResFW, since the abstract interface toward the HMI is already provided and possibly used by existing HMIs.

### 3.3.1 The navigation system Navit

At the time of writing there is a resource in ResFW for navigation, the open-source project Navit[6], slightly modified for the needs of Pelagicore. It is treated in a special way however – it is not a resource in ResFW. Instead, it can be accessed by anyone over D-bus.

Navit allows navigation to locations specified by longitude and latitude coordinates. It also has the possibility to set and get the user destination, as well as set (but not get) the user position when simulating driving to a destination. Some interesting features such as showing specific points of interest or adding via-locations to a navigation, are not supported in the Pelagicore version of Navit. All points of interest are shown on the display constantly, but without any functionality. The only way to use a point of interest is to spot it when driving and then try to navigate there manually.

Communication with Navit is done over D-bus, providing an interface to most of the functionality of Navit. Plain navigation can be started by providing a destination in longitude and latitude coordinates.

For the user, Navit shows a map, names of places and some icons for place categories (see figure 3.5). All of this information, along with the map data, such as roads and streets, buildings, forests, et cetera, are taken directly from the Navit database. Navit can be configured for several different databases but the Pelagicore version uses OpenStreetMap[7].

## 3.4 Pelagicore HMI

Pelagicore have a reference human-machine interface (HMI, see figure 3.6) connected to ResFW, used to demonstrate its abilities. Normally, the HMI is developed by a customer, designing their own look and feel but using an existing API to resources – ResFW. In the thesis, the Pelagicore reference HMI will be used. It is basically a graphical user interface, commonly used with touch screens and some physical buttons to provide input.

To enable HMI developers to easily implement their graphical design, fast and across platforms, the C++ framework Qt[8] is used to interface ResFW. The graphical HMI design is specified in the Qt markup language (QML), connecting to a layer of Qt C++ code beneath, that in turn communicates with ResFW over D-bus. The HMI can set up

[6]Navit is a car navigation system with routing engine. See the website at: `http://www.navit-project.org` [Accessed July 9th, 2012]

[7]OpenStreetMap is a database with open-source map data. See the wiki at: `http://wiki.openstreetmap.org` [Accessed July 9th, 2012]

[8]Qt is a development framework to create applications and graphical user interfaces for desktop, embedded and mobile platforms. See their website at: `http://qt.digia.com/` [Accessed Oct 25th, 2012]

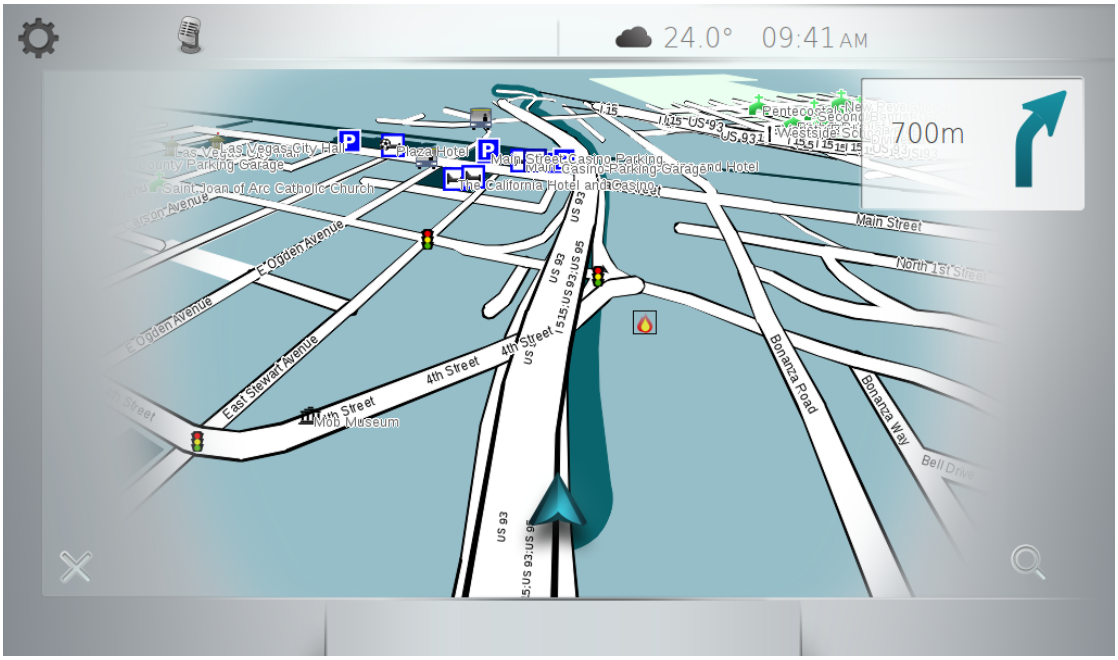



**Figure 3.5:** Navit displays its map in the Pelagicore HMI. The map shows streets, buildings, forests, names of places and some icons for place categories.

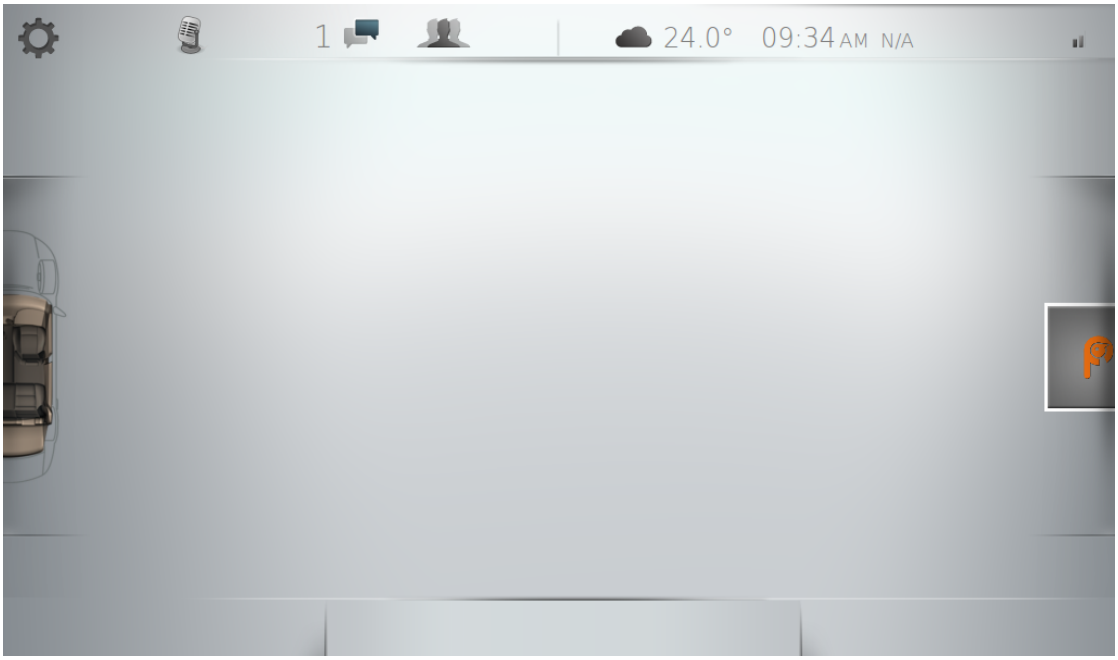



**Figure 3.6:** This is the Pelagicore reference HMI, showing the start view. Swiping upwards on the touch screen brings up the Navit map.

subscriptions for D-bus messages and directly send D-bus messages to request resource data. Thus, the HMI can access any resources without ResFW ever knowing about the existance of the HMI. ResFW just sends data blindly as it is requested.

# 4

# First Prototype – Static Demonstrator

The static demonstrator is the first functional version of the Pelagicore In-Vehicle Infotainment system with spoken dialogue control. It integrates the Pelagicore ResFW and HMI with Talkamatic Dialogue Manager (TDM) for spoken dialogue (SD) control and Navit for navigation. The grammar is written specifically for navigation and weather, focusing on named places. In the static demonstrator, around twenty named places are supported. The idea is to support any place around the world, but the static demonstrator cannot look up new ones. Of course, this critically limits the usefulness of the system but this is again the first prototype. Support for places across the world will come with the second prototype, the dynamic demonstrator, which will be discussed in chapter 5.

In the static demonstrator, places are instead hard-coded in grammar. Using these places though, one can both navigate and request weather forecasts. See section 4.2.3 for some useful dialogue examples.

## 4.1 Components

As seen in figure 4.1, the spoken dialogue adapter is the heart of the static demonstrator, connecting major components such as ResFW (section 3.3), TDM (section 3.1) and Navit (section 3.3.1). The spoken dialogue adapter and its connections are described in section 4.1.1. There is also a stand-alone weather service, fetching weather forecasts upon requests through ResFW. Its details are described in section 4.1.2 below.

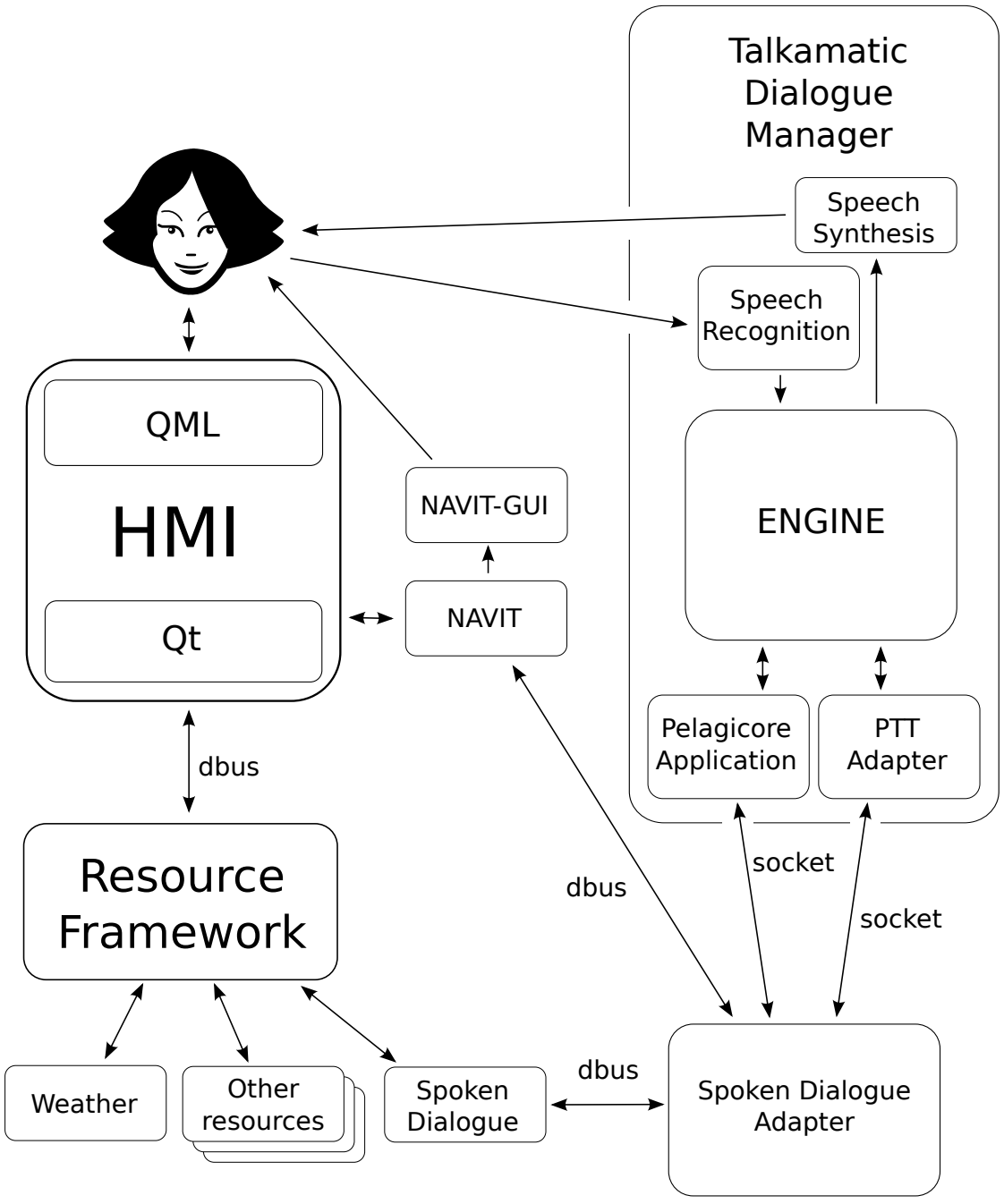



**Figure 4.1:** The static demonstrator integrates Talkamatic Dialogue Manager (TDM), Pelagicore Resource Framework (ResFW) and Navit navigation with the Pelagicore human machine interface (HMI) through the Spoken Dialogue Adapter class, communicating over both D-bus and sockets.

### 4.1.1 Spoken Dialogue Adapter

As mentioned above, the static demonstrator integrates ResFW with an HMI and with TDM and Navit. This is not a trivial task and is done through an adapter class, the Spoken Dialogue Adapter. In figure 4.1, the structure of the solution is outlined. The adapter class communicates with TDM through two different sockets, one enabling push-to-talk functionality and the other communicating with the application. It also communicates with Navit and ResFW via D-bus. See below for explanations of each connection respectively.

#### 4.1.1.1 The TDM PTT Connection

The PTT connection enables the use of a push-to-talk (PTT) button. Through the ResFW-connection, PTT messages are then passed on to the HMI, allowing it to display a graphical notification when the user presses the PTT button. In the same way, a PTT button can be shown in the HMI, which upon a press sends a signal through the adapter to TDM, to start listening to user speech.

#### 4.1.1.2 The TDM Application Connection

When the user makes a request, for example to start navigation to a city, it is first interpreted by the automatic speech recognizer (ASR) and then by the TDM application, as described in section 3.1.4. If the interpretation goes well, the TDM application tells the adapter to service the request. The request is then passed on to either ResFW or Navit, depending on what to do. If the interpretation for some reason does not go well, the user will be told and has to repeat her utterance.

The connection between the TDM application and the adapter is based on a socket, using an XML protocol to specify communication (see appendix A). In the static demonstrator, the communication is unidirectional from the TDM application to the adapter, consisting of two messages. One of the messages is to start navigation, the other to show a weather forecast.

#### 4.1.1.3 The Navit connection

Navit is supposedly a navigation resource, but ended up outside of ResFW when it was integrated. See section 3.3.1 for details.

Communication with Navit is done separately over D-bus. In the static-demonstrator, some cities are hard-coded in grammar and these are the only possible destinations. To start navigation to one of them, the adapter sets the destination to the longitude and latitude coordinates of the city.

#### 4.1.1.4 The Resource Framework Connection

Communication with ResFW is done over D-bus. On the ResFW side of communications, a spoken dialogue resource appeared, providing PTT functionality to the HMI

- Provide a local cache with a duration of at least ten minutes
  - An HTTP cache listening to the header entries 'cache-control' and 'max-age' enables the caching. The weather service itself makes sure to never call yr.no with a max-age of less than ten minutes.
- Credit yr.no and the Norwegian Meteorological Insitute
  - A standard credit string is supplied with all forecasts shown in the graphical user interface.

**Figure 4.2:** These are the yr.no compliance rules. All are met in the static demonstrator.

developer. That is all that the HMI developer ever sees. To him, the entire SD system is a black box. What actually happens when the PTT function runs is that a signal travels from ResFW to the spoken dialogue adapter, which sends a PTT request on the TDM PTT connection. Eventually, a return message appears, travelling the whole way back although transforming into a D-bus signal on the way. The result confirms the PTT request, letting the user know that TDM and the ASR is listening.

Finally, when the user finishes speaking, the ASR considers the speech finished and sends a new return message, letting the user know that it have stopped listening. This message travels the same route as the first PTT confirmation message.

### 4.1.2 Weather Service

The weather service is a separate program developed to take care of weather requests, delivering forecasts in a known format. It is part of ResFW, fitting well with a D-bus interface and known output. Internally, the yr.no The Norwegian Meteorological Institute (2012b) web service for location forecasts is used to deliver forecasts. yr.no however has some rules (The Norwegian Meteorological Institute, 2012a) that must be met in order to use their web service. The weather service I implemented here complies with all of them. See figure 4.2 for the relevant rules and how they are met. Note that results from yr.no are delivered in XML format but converted to a ResFW-specific format before they are sent along across D-bus.

### 4.1.3 Human-Machine Interface

When requesting a weather forecast through ResFW, the forecast is always sent to the human-machine interface (HMI). It is up to the HMI itself to decide whether the forecast should be shown or not, rather than letting the resource have that kind of control. This means that any other resource, not just the spoken dialogue one, can request a forecast and have it shown in the HMI. Currently, the HMI has one forecast display, showing four days of weather. The current day's weather is shown in the top half, including the place name. Then the weather for the coming three days is shown separately in the bottom

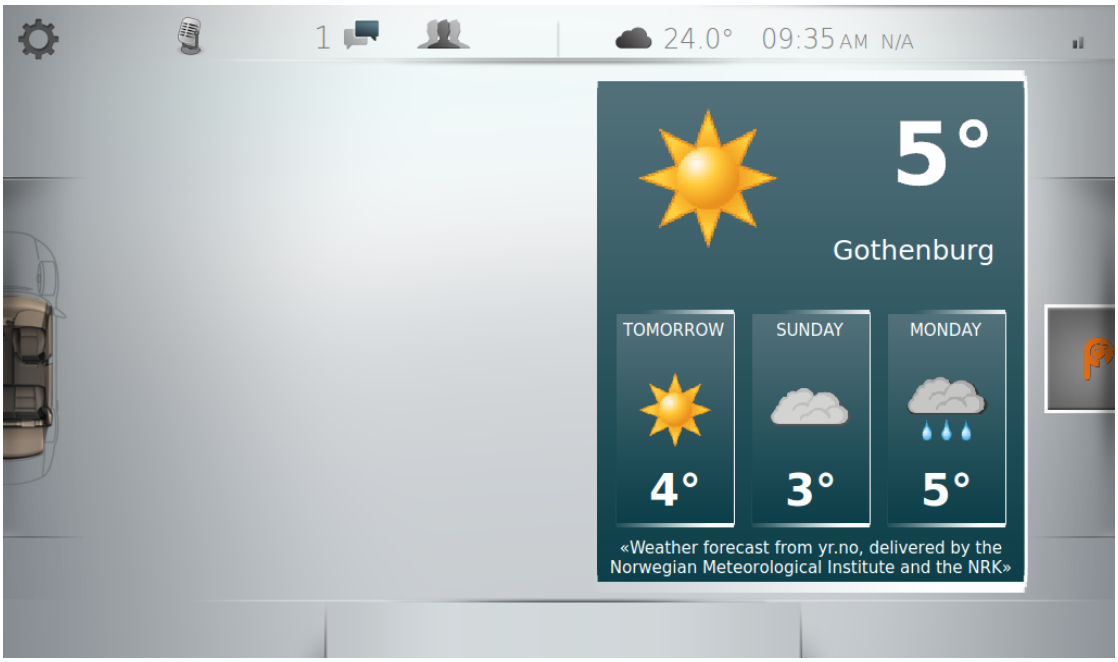



**Figure 4.3:** Weather forecasts are shown in a window on the right side of the HMI. Today's weather is shown in the top half while the coming three days are shown below. For each day there is a symbol, showing what kind of weather to expect, and the temperature in written numbers. The forecast location is also displayed, in this case Gothenburg.

half. See figure 4.3 for a picture.

## 4.2 TDM Application

The Talkamatic Dialogue Manager (TDM) application holds everything specific to the implemented use cases and the adaption to the domain. See section 3.1 for an introduction to TDM and section 3.1.4 to learn about its applications and what they contain. Use cases in the TDM application of the static demonstrator focus on places, of two different kinds. First there are geographical locations, such as cities. Second, there are named entities, such as restaurants and gas stations. Read more about them and how they are organized in section 4.2.2 below. The use cases themselves are described in section 4.2.1.

### 4.2.1 Use Case Tasks and Plans

Early in the thesis I made a list of desired use case tasks, relevant to the two kinds of places mentioned above. I also made a list of named entity categories, relevant to navigation.

Both lists are appended, see appendix E. The use case tasks chosen from the list are listed below, with the plans that use them described in the sections that follow.

```
ISSUE : nav_to
PLAN:
  findout(?x.nav_to_place(x))
  dev_perform(?x.nav_to(x))
```

**Figure 4.4:** The nav_to plan starts a new navigation in the HMI. The nav_to_place is the destination.

```
ISSUE : show_weather
PLAN:
  findout(?x.show_weather_place(x))
  findout(?x.show_weather_time(x))
  dev_perform(?x.show_weather(x))
```

**Figure 4.5:** The show_weather plan shows a weather forecast in the HMI. The show_weather_place specifies the location and the show_weather_time specifies a week day for the forecast.

1. Navigate to a location
2. Navigate to a named entity at, or near, a location
3. Display weather at, or near, location or named entity
4. Display weather at a specified time or within a specified timespan

#### nav_to

The first two use cases, navigate to location and named entity respectively, are implemented in the same plan in the TDM application – the `nav_to` plan (figure 4.4). Locations and named entities are handled in the same way by the application, both regarded as places. See an example of how both kinds can be used in the same way in figure 4.7.

#### show_weather

The remaining two use cases, concerning display of weather, are also implemented in the same plan in the TDM application – the `show_weather` plan (figure 4.5). They are combinable so that the user can specify both place and time (in this case time is a weekday) in the same utterance. If the user does not specify a place however, the application uses the current user position. Similarly, if the user does not specify a time, the application uses the current time.

```
ISSUE : nav_do
PLAN:
  findout(?x.nav_do_verb(x))
  findout(?x.nav_do_place(x))
  dev_perform(?x.nav_do(x))
```

**Figure 4.6:** The nav_do plan starts a navigation. If the user does not specify a nav_do_place, the static demonstrator will find one for her. It uses the nav_do_verb and selects a place where the user can do what the verb describes.

#### nav_do

In addition to the above mentioned plans, another one has been implemented, the `nav_do` plan (figure 4.6). It results in the second use case in the list above, navigation to named entities. The idea with this plan is to help a user that wants to do something. It helps the static demonstrator interpret verbs related to places. Or rather interpret what the user wants to do and find a named entity where she can do such a thing, also starting navigation there. For example, eating can be done at restaurants and cafes. For a better description and examples of what this plan can do, see figure 4.9.

### 4.2.2 Data

In the static demonstrator, data is totally static, it is hard-coded and there are only around twenty places. Places are of the following types:

- **Locations** – Geographical locations, for example countries, cities, neighborhoods. USA, New York, Brooklyn are all geographical locations.
- **Named Entities** – For example specific restaurants, gas stations, parking lots or schools, as long as they have a name. Starbucks, Walmart and New York University are all named entities.

Place data is located in data files on disk, mapping places to place data. Some data entries are required, some optional.

- **Required place data** – Used by both locations and named entities.
  - **English name** – Each place has a name, for example `San Francisco`.
  - **longitude** – Each place has a position, specified by a coordinate in longitude and latitude.
  - **latitude** – See longitude above.
- **Optional place data** – Not used for both types of places, but necessary for named entities, for example when displaying weather forecasts in the HMI.

```
U> I want to go to San Francisco
S> Okay. Started navigation to San Francisco

U> I want to go to The Cheesecake Factory
S> Okay. Started navigation to The Cheesecake Factory
```

**Figure 4.7:** The static demonstrator is able to start navigations to places, either locations such as the city San Francisco or named entities such as The Cheesecake Factory, a restaurant in San Francisco.

```
U> I want to go to San Francisco
S> Okay. Started navigation to San Francisco
U> what will the weather be like
S> Okay. Showing the weather for San Francisco, now
```

**Figure 4.8:** The static demonstrator supports two kinds of use cases, modelled in plans. The two kinds are navigation and weather and both of them focus on places. The static demonstrator is able to keep places in context, between plans.

- **category** – The category of a named entity. Locations does not have this. A list of desired categories is appended in appendix E.2, although only two are supported in the static demonstrator – `Food & Drink` and `Gas station`.
- **address** – The address to a named entity. Locations does not have this as they cover larger areas than can be specified with an address.

### 4.2.3 Dialogue

The static demonstrator dialogue focuses on places, both locations and named entities. But how does it work? To describe the different kinds of places, I will give some examples of how the dialogue could look in the static demonstrator. Figure 4.7 shows how the static demonstrator can interpret both a city (location) and a named restaurant as the same thing – places. Internally, the static demonstrator had knowledge about which type a place had, although to the user, it all seemed coherent.

The static demonstrator can also keep places in context between plans. See the example in figure 4.8. The static demonstrator is told about the a location, `San Francisco`, in the first request. The `nav_to` plan is used to start a navigation. The second request is to show the weather, but no place is specified so the static demonstrator uses the place in context, `San Francisco`. This time, the `show_weather` plan is used to show the weather in the HMI. Keeping places in context also works the other way round, with weather first and navigation after.

Three different scenarios are demonstrated in figure 4.9. The first scenario covers the case where the user mentions both a verb and a place. In that case the static demonstrator has several options since it has knowledge about both the place and the

```
(A)
U> I want to eat at The Cheesecake Factory
S> Okay. Started navigation to The Cheesecake Factory

(B)
U> I want to eat at Auto City
S> Okay. Started navigation to Auto City

(C)
U> I want to eat
S> Okay. Started navigation to The Cheesecake Factory
```

**Figure 4.9:** The static demonstrator has knowledge about verbs such as eat or refuel. If the user tells the static demonstrator that she wants to eat, the demonstrator finds her a place where she can eat and starts a navigation there.

verb. In `(A)`, the place is a restaurant and everything is fine. However in `(B)`, the place is a gas station and the static demonstrator could tell the user that:

```
"no, you can not eat there, choose another place"
```

But perhaps the user has local knowledge and knows that in fact

```
"yes, actually you can!"
```

So the static demonstrator takes a passive stance instead and simply starts a navigation to the place, no matter what the user wants to do there. In `(C)`, the user does not specify a place. Instead, based on the user position, the place in context and an ongoing navigation destination, the static demonstrator selects a suitable place. In this case it is the same restaurant as in the first scenario.

## 4.3 Shortcomings

Some shortcomings exist in the static demonstrator. Some relate to the purpose (see section 1.1) of the thesis and some relate to an unfinished and unpolished implementation.

### 4.3.1 Static Grammars

The shortcomings relating to the thesis purpose are mostly due to the static grammars of the static demonstrator implementation. Of course, that is why it is called *static* in the first place.

Names of places are hard-coded and around twenty in numbers, so knowledge of places around the world is very limited. Categories of named entities are also hard-coded, which in itself is reasonable since there exist a finite number of categories. The static demonstrator only supports two of them though.

```
U> I want to eat at The Cheesecake Factory
S> Okay. Started navigation to The Cheesecake Factory

U> show me the weather
S> Okay. Showing the weather for The Cheesecake Factory, now
```

**Figure 4.10:** The user wants to show the weather for her destination, The Cheesecake Factory. In the context of weather forecasts, one may consider an individual named entity rather small. One may wish that the name of its geographical location should be mentioned instead, or at least in addition to the named entity.

Both of these shortcomings are mentioned in section 4.2.2 above. Additionally, the grammar itself only covers so many utterances. Until thorough user tests can be performed it is unknown how restricted the grammar is.

#### Proposed Solution

A solution to the problem with static grammars is to provide the demonstrator with world knowledge. With the case of hard-coded place names, they may be extended to cover all the names in the world and loaded on startup. This still involves a static grammar, which seems to be a common approach among SD systems in in-vehicle infotainment (IVI) systems at the time of the thesis (see section 2.4). No modern IVI SD systems support names though, probably because of their use of static grammars (see section 2.4). Loading names on startup would create a huge list, since places include anything from geographical locations to named entities of many different categories. Named entities also develops over time, some disappearing, new created.

Another alternative is to dynamically load new places at run time when they prove relevant to the user. A place can be considered relevant either if the user is nearby, talks about it or perhaps navigates to it.

When it comes to categories of named entities as well as utterances, the solution is simpler. Since there is a finite number of them, they just need to be added to the demonstrator, of course also including places for each new category.

### 4.3.2 Spoken User Feedback

When the system replies to the user in speech, it is desirable to have feedback as shown in the examples above, in figures 4.7, 4.8 and 4.9. However, with the static demonstrator, feedback only uses the place in context. This occasionally results in undesired feedback, such as the one described in figure 4.10. If the user wants to show the weather for her destination, in this case a named entity, the system will use the named entity name in its feedback. A named entity may be considered rather small in a weather context. Instead, it is desirable that its geographical location is used.

#### Proposed Solution

To solve this problem, the TDM application needs to be modified. The device resource of the application can for example infer a new place from an old one, resulting in proper feedback. This solution could well have been implemented in the static demonstrator but have so far been left out.

### 4.3.3 Graphical User Feedback

Interacting with the HMI should provide graphical feedback so that the user knows that the system acknowledged the action. Similarly, speaking should provide feedback as well. Trivially, it should provide spoken feedback since this is an SD system, but also graphical feedback showing that the system acknowledges the action just like it does if it is taken through the HMI.

In the static demonstrator, graphical feedback is not always clear. As described in section 4.1.3, the `show_weather` plan always results in graphical feedback. However, the `nav_to` and `nav_do` plans just sets a new destination in Navit, not providing any really clear visual cues that something actually happens. If no navigation is active, starting a new one provides clear visual feedback, showing the route, directions, et cetera. The destination is not shown in the navigation however, so the only feedback the user gets is that she is actually navigating to the correct place, is the spoken feedback. If a navigation is started while one is already active, the user may not notice that the destination changes.

#### Proposed Solution

The solution is rather simple and requires measures to be taken to the HMI. In the navigation view, the destination can simply be displayed, changing when a new navigation is started. While the solution sounds rather simple it is slightly more complex, since Navit that keeps track of the destination is not part of ResFW, see section 3.3.1. This means that instead of just telling Navit to set a new destination, ResFW must also tell the HMI that a new destination has been set. Still, it is a rather small change with big impact on user experience.

# 5

# Second Prototype – Dynamic Demonstrator

The dynamic demonstrator is the second version of the Pelagicore In-Vehicle Infotainment (IVI) system with spoken dialogue (SD) control and the resulting prototype of the thesis. It builds upon the static demonstrator described in chapter 4 but features some critical improvements. Primarily, it tries to solve the static grammar problem described in section 4.3.1. This is done by fetching actual place data from the Internet, reloading the Talkamatic Dialogue Manager (TDM) application at runtime as proposed in the just mentioned section. More details on the targeted problem are described in section 5.1. This solution moves away from the commonly used static grammars in IVI SD systems (see section 2.4). Often, named entities are only loaded on initialization, or not at all as described in section 2.4.

The new components introduced to the demonstrator for this solution are described in section 5.2 below and the changes to the TDM application are described in section 5.3. Some new shortcomings have appeared with this solution as well, explained in section 5.4 with proposed solutions.

## 5.1 New functionality

The dynamic demonstrator tries to solve the static grammar problem. From the two proposed solutions in section 4.3.1, I have chosen to dynamically load new places at run time. The TDM application is compiled at run-time with new places, then loaded by TDM.

To allow for the new functionality, new components have been constructed and some of the existing components have been modified. New and modified components are described in section 5.2 below. Changes have also been made to the TDM application, described in section 5.3 below.

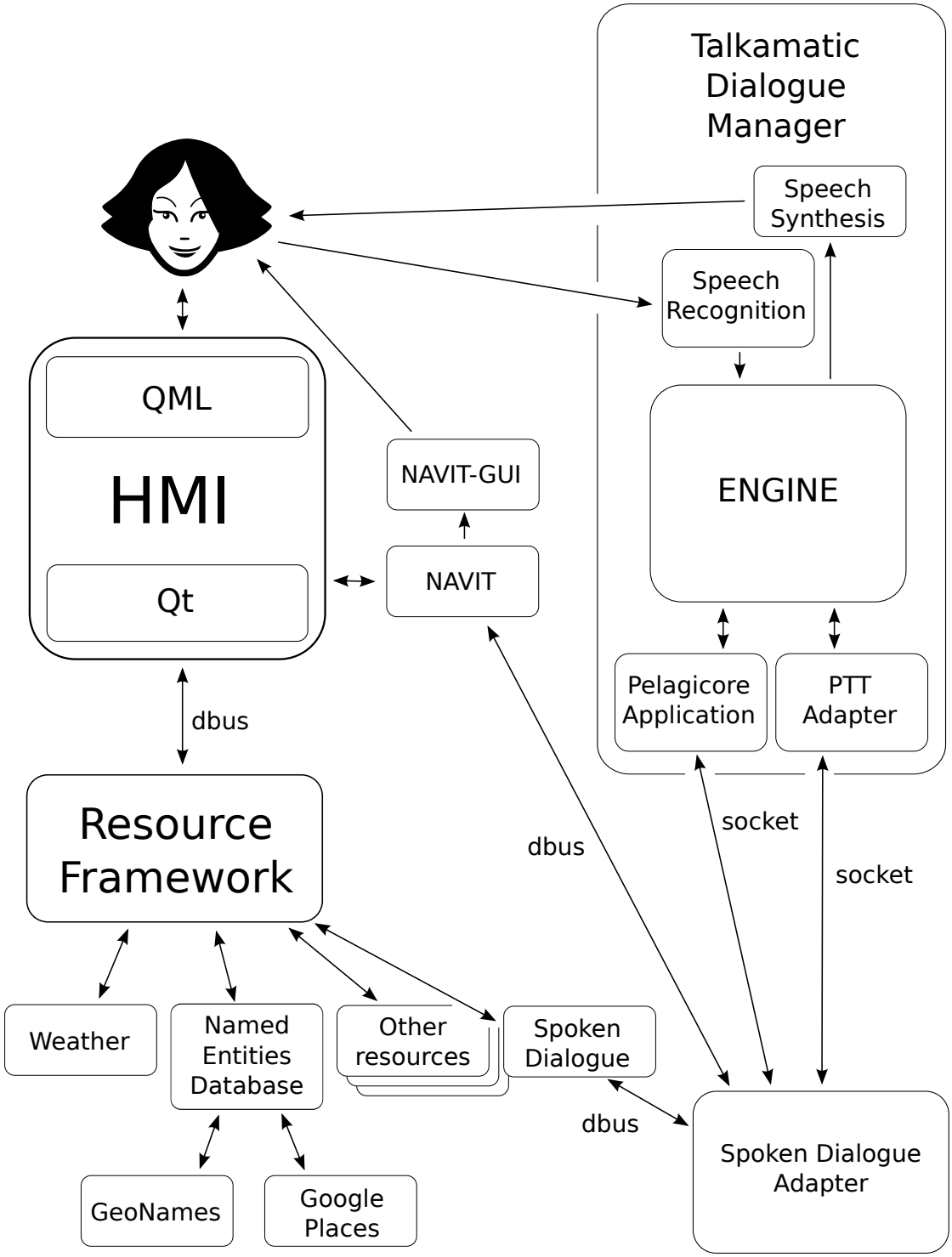



**Figure 5.1:** The dynamic demonstrator is nearly identical to the static demonstrator, but additionally contains a named entities database. It is a resource in ResFW.

## 5.2 Components

The components that had to be modified when evolving the static demonstrator are described below. The sections correspond to those of the static demonstrator components section (4.1), with two exceptions. Firstly, a data compilation module have been added to the spoken dialogue adapter, compiling new data for TDM as mentioned in section 3.1.4. Secondly, an entirely new component has been added, the named entities database, (see section 5.2.2 below). Just like the weather service (section 4.1.2), requests to the named entities database goes through the Pelagicore resource framework (ResFW).

### 5.2.1 Spoken Dialogue Adapter

The biggest difference to the spoken dialogue adapter, is bidirectional communication with the TDM application. This is needed to reply to TDM application requests over the TDM application connection. Several new messages have also been added to the TDM application protocol (see appendix A). The replies themselves have been added but also messages to allow the demonstrator to work dynamically, for example data compilation requests (see section 5.2.1.1). Additionally, requests for new places can be made to the named entities database (section 5.2.1.4).

#### 5.2.1.1 Compiling Places

The TDM application needs to be compiled before it can be initialized, so it needs to be compiled and reloaded at run-time in the dynamic demonstrator. The different resources of the TDM application need to access place data, but on slightly different levels. So, if the place data change, then the application needs to be compiled anew, providing place data on all levels. These levels are enabled by two extra compilation steps – the place exporter and the place compiler.

**Place Exporter**

The place exporter is written for specifically for the dynamic demonstrator to support dynamic places, for both geographical locations and named entities. It keeps locations and named entities separated and writes them to the place ontology, a resource that can be accessed by the place compiler.

**Place Compiler**

The place compiler is also written with the sole purpose of supporting dynamic places. It converts the locations and named entities of the place ontology to places – without any notion of place types. Places, in this sense, are a part of the TDM application, which can be compiled by the regular TDM compiler (mentioned in section 3.1.4).

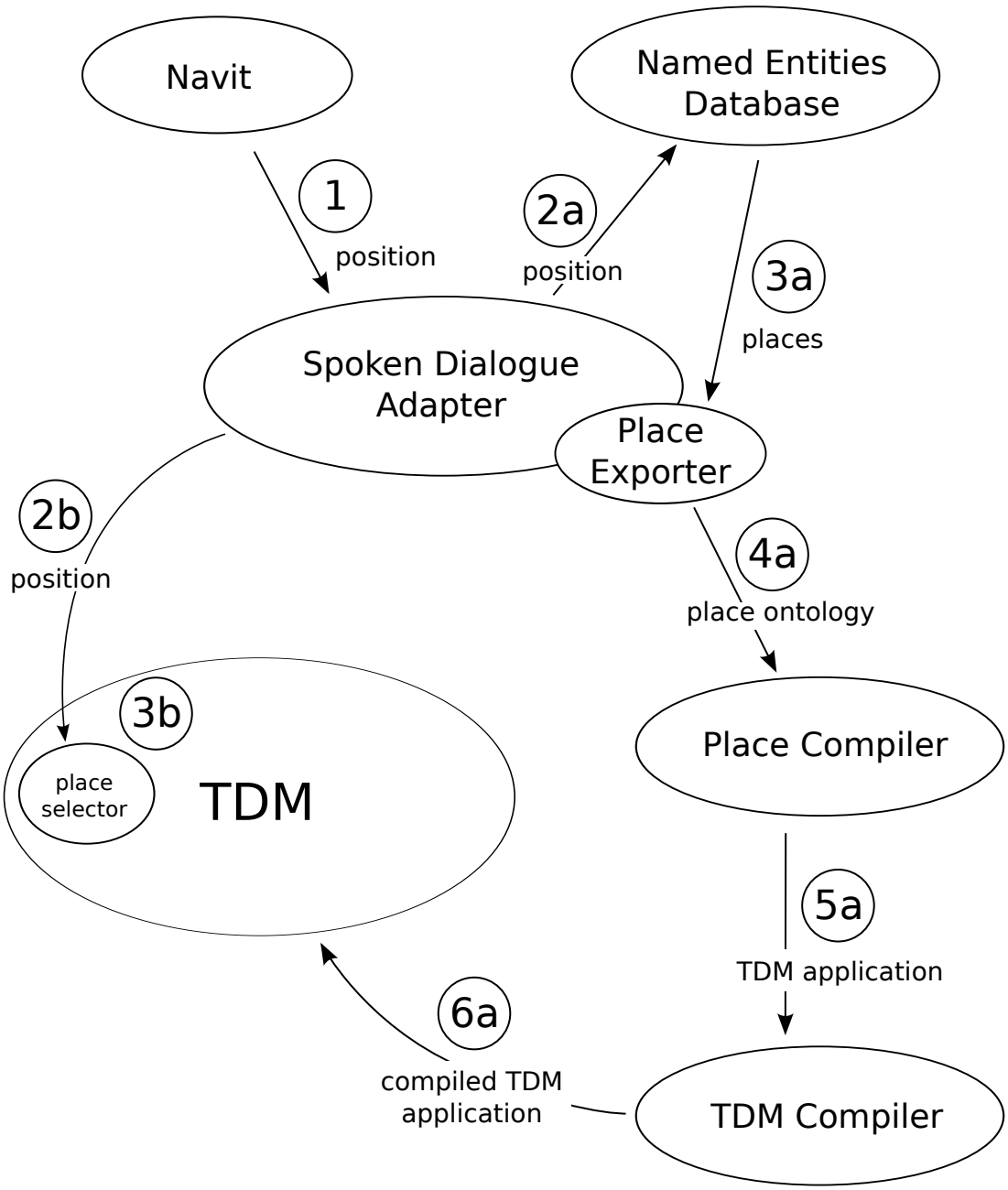



**Figure 5.2:** The flow of actions that are related to places and to compilation are shown in this figure. Navit handles position requests, returning the user position to the spoken dialogue adapter (1). There can be two reasons for a position request. Either (a), a compilation request, passing the position to the named entities database (2a) to compile new places (3a); or (b), place selection (2b, 3b). In the compilation sequence, places are exported to the place ontology (4a). The place compiler compiles places for the TDM application, updating the old application with new places (5a). In step 6a the regular TDM compiler kicks in.

#### 5.2.1.2 The TDM Application Connection

Since the TDM application in the dynamic demonstrator receives replies from the spoken dialogue adapter, the TDM application connection supports this. The solution extends the protocol, with several new message added to it. The full list of messages can be found in appendix A.

Unfortunately, there is a limitation in the data request message used to reload places. Only one named entity category can be given in each request. This was initially supposed to be used when the category is known so that only requests with one named entity category are needed. It is however impossible to know what category the user may talk about in the future, so all named entity categories need to be loaded at all times. The data request message still only supports one category though. To the user, this means that she may only talk about one particular verb in the `nav_do` plan (section 4.2.1) – `eat`.

#### 5.2.1.3 The Navit Connection

From the original Navit resource it was not possible to get the user position, as mentioned in section 3.3.1. The TDM application however needs to know the user position. Meanwhile, Navit is the natural component to ask for such a thing, being used as source for everything related to the geographical world.

The implementation is a bit tricky. It is done in the Pelagicore version of Navit, which is a slightly modified version of an old snapshot of Navit.

Navit does not seem to have a representation of user postion though, so instead I used the map center, which is the longitude and latitude coordinate that is centered on the screen. It is a solution specific to Pelagicore, but it works thanks to the fact that Pelagicore always keep the map centered on the user position anyway.

This solution can be exchanged for a better one, once one is provided.

#### 5.2.1.4 The Resource Framework Connection

The ResFW connection have been modified as well since there is now a new resource in ResFW used by the spoken dialogue adapter, the named entities database (section 5.2.2). While doing this change, I also changed how the weather service (section 4.1.2) is used. In the static demonstrator, the spoken dialogue resource in ResFW (mentioned in section 4.1.1.4) took care of weather requests. In the dynamic demonstrator however, the spoken dialogue resource in ResFW supplies the spoken dialogue adapter with agents for the weather resource and the named entities database resource respectively (both are ResFW resources). An agent in this context means an interface through which the spoken dialogue adapter can make relevant requests and receive their results.

From the ResFW point of view, the agents are represented by objects wrapped in the actual resources. Since resources are interfaced through D-bus, so are the agents. They have unique D-bus object paths and provide interfaces specific to the purpose of receiving requests from the spoken dialogue adapter.

### 5.2.2 Named Entities Database

The named entities database is, similarly to the weather service, a separate program. It handles requests for locations and named entities close to a position. Through the provided interface the user can request either locations or named entities of a specific category, within a specific radius of a longitude and latitude coordinate. To handle requests for locations and named entities in an optimal way, different web services are used, one for locations and one for named entities. The services deliver their data in different formats so they are converted to a common format in the ResFW resource, see section 5.2.2.3 below.

#### 5.2.2.1 The Locations Web Service

GeoNames[1] is used for finding locations. It covers over eight million geographical locations across the world, provided through a web API.

By giving GeoNames a bounding box it returns a suitable amount of results from inside the box. The results returned are based on importance, so the bigger the bounding box, the bigger and more important the results returned. Covering the whole world for example would result in some countries and some capitals. Usually, the number of locations returned are between 10 and 20.

The bounding box is rectangular but since I use a circle radius to specify it, a conversion is made. The bounding box is calculated as the smallest square wrapping the circle.

#### 5.2.2.2 The Named Entities Web Service

Google Places[2] is used for finding named entities. It covers almost any named entity category such as restaurants, barber shops, parking lots, et cetera, all over the world. Not to mention it provides details such as address, formatted address, phone number, postal code, et cetera. This implementation makes use of name, longitude and latitude coordinate, address and category.

Google Places use the specified radius directly, returning the most important results within the radius of the position. For each request, 20 results are returned with the option to get another 20 by sending a new request, passing a token found in the first result; a maximum of 60 results can be returned in total.

It is also possible to specify categories in the request, on a very fine level. There are for example several different categories of where you can eat, for example `cafe`, `food` and `restaurant` (Google, 2012).

[1] GeoNames is a web service for finding location names across the world, plus much more. Website: `http://www.geonames.org/` [Accessed Oct 4th, 2012]

[2] The Google Places API is documented online, at `https://developers.google.com/places/documentation/` [Accessed Oct 4th, 2012]

- Required place data entries
  - Name
  - Longitude
  - Latitude
- Optional place data entries (only valid for named entities)
  - Address
  - Named Entity Category

**Figure 5.3:** Required place data is needed for both locations and named entities while optional place data is only needed for named entities

#### 5.2.2.3 Place Data

The two web services for places return results in their own ways. To better support the concept of ResFW resources wrapping several services of the same type in the same way, the data is converted to a common format in the named entities database resource in ResFW. This also supports the notion of locations and named entities both being places.

Essential to places is of course the longitude and latitude coordinate but also the name of the place. These data entries are considered required. There are however two additional data entries provided for named entities: address and category. These are considered optional. Figure 5.3 summarizes it.

## 5.3 TDM Application

The TDM application (section 3.1.4) remains mostly the same in the dynamic demonstrator compared to the static. All modifications that have been implemented are mentioned in this section.

Mainly one difference exists between the demonstrators. The dynamic demonstrator requires the TDM application to be able to receive data from the spoken dialogue adapter. For example, it needs knowledge about the user position and destination. The reason that it needs this information is that the TDM application identifies places by name, while the spoken dialogue adapter gets positions as coordinates, having no knowledge of the TDM application place names.

Positions in coordinate form can be used in two ways. Firstly, the position is needed directly by the device resource to select a place, either a location representing a position or a named entity to navigate to. They are used in the `show_weather` and `nav_do` plans mentioned in section 4.2.1 and shown in the action flow diagram in figure 5.4. The place selection process is described in section 5.3.2. Secondly, the position is needed when compiling and loading new places. As seen in figure 5.2, data compilation consists of three steps. Although internal names are assigned in the first compilation step (step

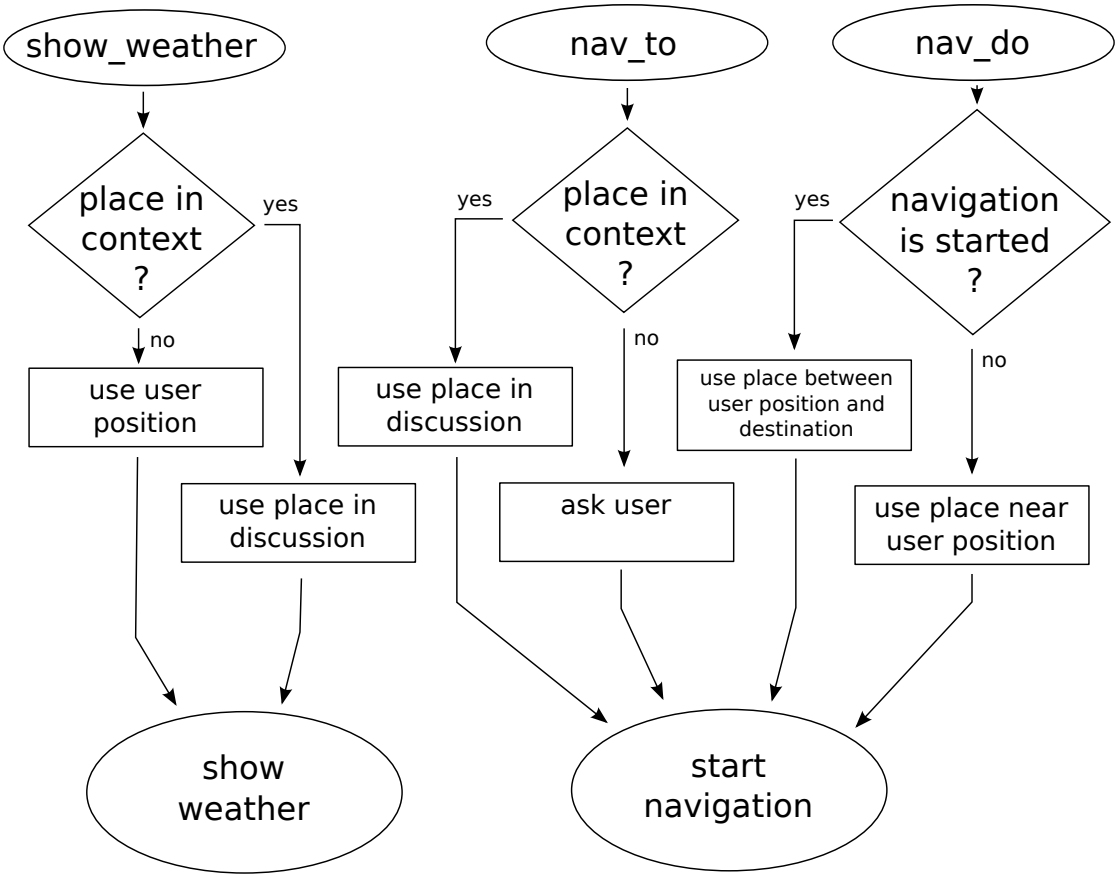



**Figure 5.4:** The flow of actions in the TDM application plans are shown here. When requesting the weather to be shown the TDM application uses either the user position or the place in discussion as the weather forecast location. Starting a navigation can be done in two ways, either through the nav_to or the nav_do plan. In the nav_to plan, TDM asks the user if there is no place in discussion. In the nav_do plan however, it is guaranteed that a category is specified, so there is no need to ask the user for a specific named entity – one can always be found either near the user position or toward the destination, in case a navigation is already started.

```
ISSUE : nav_to
PLAN:
  findout(?x.nav_to_place(x))
  dev_perform(?x.nav_to(x))
POSTPLAN:
  change_domain(domain(dynamic_demonstrator))
```

**Figure 5.5:** The nav_to plan in the dynamic demonstrator is identical to the one in the static, except for the change_domain item in the postplan. This makes sure that TDM reloads the application (called dynamic_demonstrator), which have been compiled in the dev_perform step.

```
ISSUE : show_weather
PLAN:
  findout(?x.show_weather_place(x))
  findout(?x.show_weather_time(x))
  dev_perform(?x.show_weather(x))
POSTPLAN:
  change_domain(domain(dynamic_demonstrator))
```

**Figure 5.6:** The nav_to plan in the dynamic demonstrator is identical to the one in the static, except for the change_domain item in the postplan. This makes sure that TDM reloads the application (called dynamic_demonstrator), which have been compiled in the dev_perform step.

number 4a in the figure) the compiler in the spoken dialogue adapter just writes to disk and then forgets the names, so they are ever only known by the TDM application anyway.

It is tempting to put the place selection job on the named entities database which has all the knowledge about places, but this would ruin feedback for TDM in case the place is not already loaded. If the place is not loaded it can obviously be loaded but this would cause new problems for TDM, forgetting everything about the current plan as I will describe in section 5.3.1. There are also latencies involved when loading places, causing performance issues, which I will mention in section 5.4.

### 5.3.1 Loading data

Loading data at run time requires a readily compiled TDM application. Plans have been updated for the dynamic demonstrator with a postplan (see figures 5.5, 5.6 and 5.7). The `dev_perform` item of each plan makes sure that new place data is fetched and compiled (see section 5.3.1.1). The `change_domain` then reloads the application in TDM, enabling the new places. The reason to have it in the `postplan` is that it guarantees the reload to happen after the `dev_perform`.

Loading dynamic data, that can change on disk, is not supported by TDM. The

```
ISSUE : nav_do
PLAN:
  findout(?x.nav_do_verb(x))
  findout(?x.nav_do_place(x))
  dev_perform(?x.nav_do(x))
POSTPLAN:
  change_domain(domain(dynamic_demonstrator))
```

**Figure 5.7:** The nav_to plan in the dynamic demonstrator is identical to the one in the static, except for the change_domain item in the postplan. This makes sure that TDM reloads the application (called dynamic_demonstrator), which have been compiled in the dev_perform step.

reason is that all data is imported by TDM on startup, so loading data at run time requires it to be imported anew. I have made changes to TDM to support dynamic data. Mainly by making sure that data is always read from disk when imported.

So, loading data can be performed dynamically. But the `change_domain` procedure causes TDM to forget everything it already knew. This results in TDM returning to its startup plan, which makes it ask the user what she wants to do. Since `change_domain` plan items are performed after each plan has been completed though, the problem can be considered small for the dynamic demonstrator. There is no need to remember anything for the next plan.

#### 5.3.1.1 Area to Load

In the dynamic demonstrator, places can be loaded for one position at a time. Usually, the position is the one used in the most recent plan. It is desirable to keep places loaded for several positions, but this have not yet been implemented.

When a position is chosen to be loaded, several queries are made to the named entities database. There are two levels of queries, global and local. Basically, there can be any number of levels but the dynamic demonstrator uses these two. Both are described below, with some practical details. They are also visualized in figure 5.8.

- **Global locations** – These are used to provide knowledge about the world. Possible locations at this level can be countries, capitals, big cities and other major geographical locations. In the dynamic demonstrator, the global level covers places within a 1000km radius of the chosen position.
- **Local locations** – These provide knowledge of local locations, because locations close to the target position are more relevant than those far away of the same size. In the dynamic demonstrator, the local level covers places within a 2.5km radius of the chosen position.
- **Local named entities** – Along with local locations, named entities in the same area are also loaded.

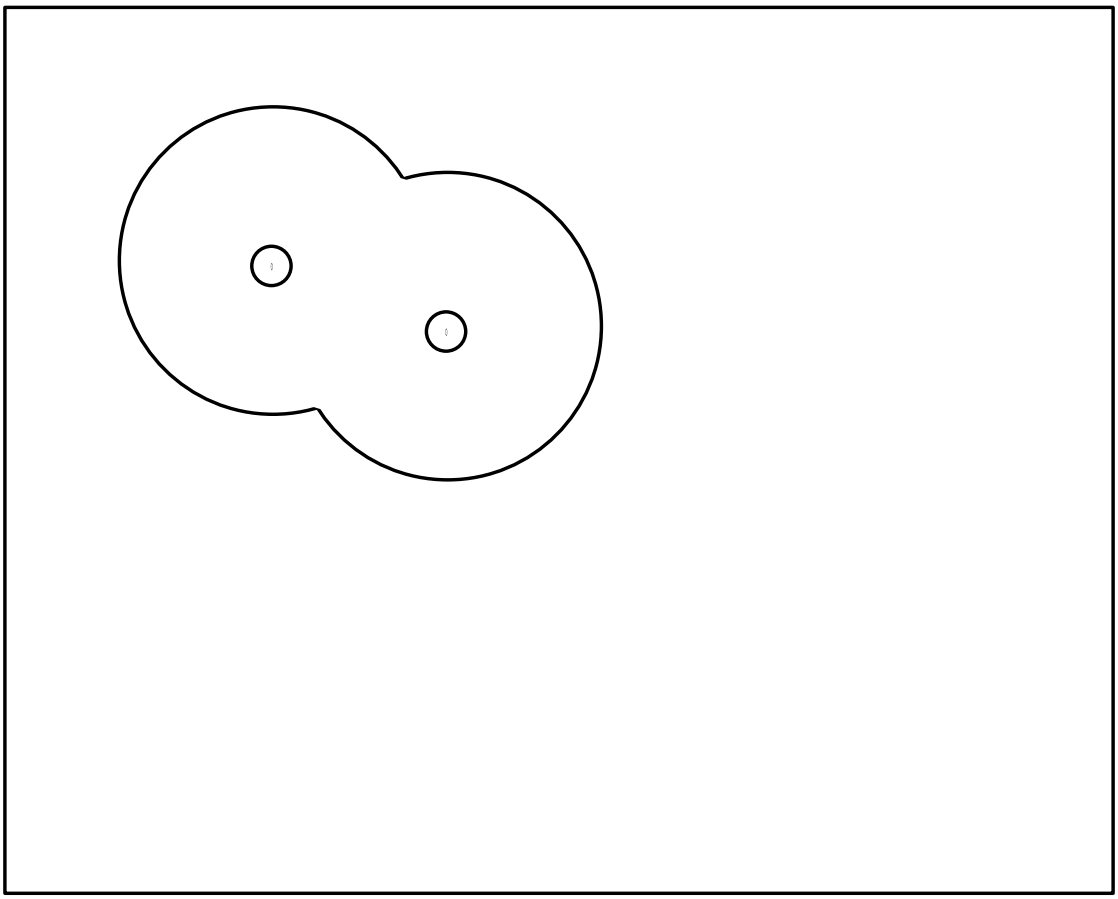


**Figure 5.8:** Places are loaded on different levels. In the dynamic demonstrator they can be loaded for one position at a time but the idea is to keep places loaded aruond all relevant positions. This figure shows how two positions are used. Locations are loaded on both a local (small circle) and global (large circle) level, while named entities are only loaded on the local level.

Unfortunately, the fact that data is only loaded for one position at a time is a big restriction, causing problems. For example, place names can be removed from TDM although they are still relevant to the dialogue or have been used in it.

It is worth saying that the idea is to always keep relevant positions loaded. For example when a navigation is started to a location, this location should remain loaded until it is no longer used. It can be considered not used if the user reaches it or if she sets a new destination. So, if the user reaches her destination, the idea is that it is no longer relevant as a destination. But it is now her position, hence relevant anyway. To conclude, the idea is that all relevant places should always be loaded in the TDM application.

### 5.3.2 Selecting places

The TDM application knows about place names. The spoken dialogue adapter does not. So when the device in the TDM application communicates with the spoken dialogue adapter, for example when requesting the position of the user, it just receives a world position as a longitude/latitude-coordinate.

To use the position within the application, the device needs to translate the coordinate to a place name given the currently loaded names. Two different selection methods have been constructed. The first is to select a named entity or location near a position, for example the user position in the `show_weather` and `nav_do` plans. The other is to select a named entity along the shortest path between two positions. For example between the user position and destination when looking for a named entity in the `nav_do` plan with navigation started (see figure 5.4).

#### Selection of place near a position

Given a position in longitude and latitude as described above, a place needs to be found. This is done by first filtering out valid places (which only applies for named entities, where they need to match a category). Next, the actual selection step, the distance is measured between the user position and every valid place, selecting the one closest to the user position.

#### Selection of place between two positions

Given two positions of longitude and latitude, a place needs to be found. The process is similar to the one of place selection near a single position, but instead of measuring distance between the position and candidate place it is now measured going from the first position, via the candidate place, to the second position.

The navigation resource of ResFW should be able to calculate distances between positions. With Navit this is not possible, at least not using the D-bus interface. Instead, distance is calculated naively as the crow flies. This resulted in a very fast implementation.

### 5.3.3 Two way communication

Communication in the TDM application is done in the device resource, sending a message across a socket using a protocol (section 5.2.1.2). When a request is sent from the device resource, a result is eventually expected back. The spoken dialogue adapter potentially has to forward the request to other systems though, so it is processed asynchronously on that end. TDM however needs the result to be returned synchronously. The device resource thus blocks the call until it the result is returned.

The blocking mechanism is implemented by letting two threads, a writer and a reader, deal with the socket communication. These threads communicate with the device through a thread-safe queue each, the reader filling the read queue with whatever comes across the socket and the writer forwarding messages on the write queue to the socket.

This way, the device can do a blocking read on the read queue until the expected result is received.

## 5.4 Shortcomings

The dynamic demonstrator (partly) solves the static grammar problem (see section 4.3.1) of the static demonstrator. The other shortcomings still remain though.

Some new shortcomings are introduced as well, described in the sections below.

### 5.4.1 Performance

With the dynamic data compilation step comes delays. They are particularly caused by the TDM compiler mentioned in section 3.1.4, corresponding to the last compilation step (6a) shown in figure 5.2. The reason behind the latency is mainly the ASR specific compiler. For example, the Nuance VoCon Hybrid ASR compiles a binary grammar, usually needing five to thirty seconds to compile roughly 40 places for the dynamic demonstrator.

#### Proposed Solution

Performance can be improved by removing the compilation step in some way. For example, Nuance ASRs have the ability to compile static parts of a grammar in advance, linking dynamic data at run-time (Nuance, 2007, p. 82). This way, most of the grammar could be compiled in advance, allowing places to be loaded dynamically. This solution however requires support to be implemented in TDM, since TDM controls the ASR.

Performance can also be sped up by running on a faster machine. My reference system is a 4GB RAM, 2x2.53GHz Intel Core2Duo CPU, single SATA HDD Apple Macmini. A server in the cloud can be used instead, taking care of the ASR while keeping the dialogue system mainly running on the client. Or perhaps the server can take care of the entire dialogue system. Both server based solutions face some technical difficulties.

### 5.4.2 Naive Place Selection

Place selection (described in section 5.3.2) is done naively in the dynamic demonstrator. Places are ranked by distance as the crow flies, disregarding all other potential ranking measures. Importance of places, fine detailed categories of named entities (mentioned in section 5.2.2), driving distance (both spatial and temporal), are all disregarded in the selection process.

#### Proposed Solution

An optimal solution would be to use all above mentioned properties when ranking places. What an implementation of this would look like is hard to tell, but would optimally be put in a single component. This component needs to have all the information readily

available. Such a component does not exist however. Navit has navigational knowledge (such as driving distances) while the named entities database has place knowledge. The solution has to be driven from the TDM application device as well, which only knows about a few places. In total, there are many issues that needs to be solved if a truly sophisticated selection process is to be implemented.

### 5.4.3 Inconsistent places

Data displayed on the map are taken from the Navit database, based on OpenStreetMap as described in section 3.3.1. Place names used in the TDM application are based on the named entities database which uses GeoNames and Google Places as described in section 5.2.2.

But what happens when the data displayed on the map does not match the data loaded in the TDM application? For example when a restaurant displayed on the map is not loaded in the TDM application. The user may say the name of the place, expecting it to be recognized. After all, it is shown on the map. But since it is not loaded in the TDM application, it will either be misinterpreted or not understood at all. In the end, the user will most likely become confused.

#### Proposed Solution

There is no clear solution to this problem. There is a great technical complexity with several co-existing systems that need to co-operate in order to solve the issue. And even if all the names shown on the map are loaded in the TDM application, what will happen when moving around? Of course, new names will become visible for the user, other names removed. Either the system needs to reload names when they come into view and go away or keep names loaded at all times. Or perhaps loading names at pre-determined boundaries instead, providing a kind of buffering.

Implementing such a solution, with requirements on both performance and availability of names, is probably not manageable with the current software design. On one hand, Navit and the TDM application uses different data sources, so name mismatches are likely; on the other hand, the TDM application requires time to reload while the map does not.

Part of a solution is to make the navigation system and the named entities database use the same database. This will solve the problem where names differ between these sources. However, it does not solve the problem of loading everything that is visible on the map. The performance issues (section 5.4.1) related to data reloads in the TDM application are limiting factors in this problem.

### 5.4.4 Inconsistent Place Names

Place names are reloaded dynamically, using the named entities database. But on every reload, a place selection is performed in the device resource of the TDM application, to get the name of the place. (sections 5.3 and 5.3.1). The name is used for example in

spoken feedback to the user. Because the named entities database is not guaranteed to always return the same results, it can happen that the name of a place suddenly changes, although the coordinates identifying the place do not.

This can only happen when a name needs to be carried along across reloads though, for example when starting navigation to a place. If later referring to the destination, the name may be missing because the named entities database did not return the actual destination in its results.

#### Proposed Solution

Adding all relevant places on every reload, instead of relying solely on the named entities database, would ensure consistent place names at all times. This would however require adjustments to the reloading process as well as the TDM application connection of the spoken dialogue adapter.

### 5.4.5 Number of Places Loaded

When loading places, all places returned by the named entities database are used. They are however a rather small number on each level (see sections 5.2.2 and 5.3.1.1). Figure 5.8 shows how places are intended to be loaded on the different levels.

The problem is that so few places are loaded on each level that they may be spread thinner than desired. The graphical coverage is simply not satisfactory. If unlucky (with 10 locations loaded globally), the smallest distance to a global location may for example be 100km (which is not unlikely with 10 locations spread between 0 and 1000km from the target position). Meanwhile, the largest distance to a local location can be 2.5km. This results in a gap of 97.5km. Such a gap can be acceptable if far enough away from the position, but currently the gap is often too large and too close to the position.

Global locations in the dynamic demonstrator are not all that global either, since they only cover the area within 1000km from the target position. There is no knowledge at all about places further away than 1000km. These facts mean that there are big gaps in the overall graphical coverage of the world.

#### Proposed Solution

Analyses can be made to measure coverage. Place requests can then be adjusted to increase coverage, weighing it against the resulting performance reduction. Of course, both graphical coverage and performance need to be satisfactory to have a usable product. User tests are ultimately needed to measure the overall usability in the product.

### 5.4.6 Number of Named Entity Categories

The dynamic demonstrator, like the static demonstrator, only supports two hard coded named entity categories. In addition to this, the dynamic demonstrator only supports loading places from one category on each reload. The reason is the protocol limitation for

the TDM application connection in the spoken dialogue adapter, mentioned in section 5.2.1.2.

#### Proposed Solution

The solution is to update the protocol of the TDM application connection in the spoken dialogue adapter to handle all possible categories. Perhaps by providing the protocol with a list of categories in a well specified format instead of just one category.

### 5.4.7 Resolution of Named Entity Categories

Named entity categories in the dynamic demonstrator are hard coded. Only two categories are supported, `Food & Drink` and `Gas station`, the same as in the static demonstrator. Especially the `Food & Drink` category can be compared to the extensive number of food-relevant categories in use in the named entities database, described in section 5.2.2.2.

This means that, if the user tells the demonstrator that she wants to eat somewhere, a place is selected that fits any of the food-relevant categories in the named entities database. It may not be at all what the user expects. Maybe a cafe is selected but the user desired a fine dining restaurant. Maybe a fine dining restaurant is selected when the user wanted take away food from a drive-in.

#### Proposed Solution

The interaction between user and demonstrator may look the same, but before finding the place the demonstrator can ask the user to specify her requirements further. The risk is that the interaction is dominated by questions, but the user should of course be able to specify a more specific category from the start, rending the questions unnecessary. Another approach is to let the user specify category if she wants to, otherwise use a default. Both solutions can be implemented directly in the TDM application by, without heading into details, modifying the ontology, domain, device and grammar resources.

# 6

# System Evaluation

This chapter describes a brief evaluation, performed to estimate the impact of the dynamic demonstrator shortcomings (chapter 5) from the user's perspective.

The evaluation attempted to estimate user satisfaction, partly by estimating the relevant factors of task completion and concept accuracy mentioned in section 2.3.1. Interviews were carried out as well, after each test, trying to judge which of the shortcomings in the demonstrator that had the biggest impact on user experience.

I should mention that concepts mentioned here correspond to dialogue moves when talking about Talkamatic Dialogue Manager (TDM).

## 6.1 User Tests

Six people (the test subjects) were selected to test the dynamic demonstrator. All six worked at the Pelagicore office, having a background in in-vehicle infotainment development but with little or no knowledge of spoken dialogue systems. All test subjects were males, five of them with an age ranging from 30-35, the sixth was in the age span of 45-50. So, the test subjects did not have a wide demographic spread. This is generally not a good idea in testing but becomes more important the bigger the test and the closer to release it is. For such a small and early test as this, the impact of demography is highly limited, but should be noted.

The test subjects were given four tasks (see appendix F) that they needed to solve. To help them get started, they were initially given a brief description of the capabilities of the demonstrator. In addition to this they were also guided along the test, in case they seemed to get stuck on utterances not supported by the demonstrator. Note that all users did the tasks in the same order, starting with number one and ending with number four.

The tasks ranged in complexity, some requiring several utterances, spanning across both use cases, some just requiring one utterance.

Measures in the tests were the ones described in sections 2.3.2 and 2.3.3, task completion and concept accuracy. To achieve high accuracy on these measures, data was gathered from audio recordings first after all tests were completed.

### 6.1.1 Results

To begin with, I would like to state that no time was available for configuration of the dynamic demonstrator and its components before these tests began. This resulted in, for example, bad speech recognition accuracy. In fact, it was lower than 50% overall. Because of this, I present two different estimations of task completion rate. The first one is based on all utterances that were required to complete tasks (*all utterances*). The second one is only based on the utterances that were recognized at all by the ASR (*successful utterances*, those that were interpreted as containing a concept). For concept accuracy, only the successful utterances were considered relevant.

Task completion rate based on all utterances is shown in figure 6.1. Note the high standard deviation on task two. This is because one troubled test subject required more than twice the utterances to complete it, compared to the others. Most of these utterances were not successful.

The task completion rate based on successful utterances is shown in figure 6.2 for each of the four tasks. In this case, the unsuccessful utterances from the troubled test subject in task two are filtered out. Thanks to this, the standard deviation and the mean for task two is significantly lower. Task four seems to be the most complex, requiring the most utterances to complete. Task three on the other hand seems to be simplest, requiring the least number of utterances. Tasks one and two are placed in the middle, task one requiring less utterances than task two.

Finally, concept accuracy is shown in figure 6.3. The accuracy is lower for tasks two and four while it is higher for tasks one and three.

## 6.2 Discussion

It is interesting to analyse and discuss the results. It is also interesting to discuss the shortcomings found during interviews with the test subjects. Both discussions are done in the below sections.

### 6.2.1 Discussion of Results

From the figures in section 6.1.1, it looks as if a more complex task (requiring more utterances to complete) leads to a lower concept accuracy. Since the concept accuracy figure only covers successful utterances though, it seems reasonable to compare it to the task completion rate figure that only covers successful utterances as well. However, there is less correlation with this figure than with the task completion rate figure that covers all utterances. Probably, the evaluation is too small to draw any statistical conclusions.

What must be mentioned again, is the low speech recognition accuracy. In roughly half of all cases, the ASR did not recognize any concept at all. This happened when

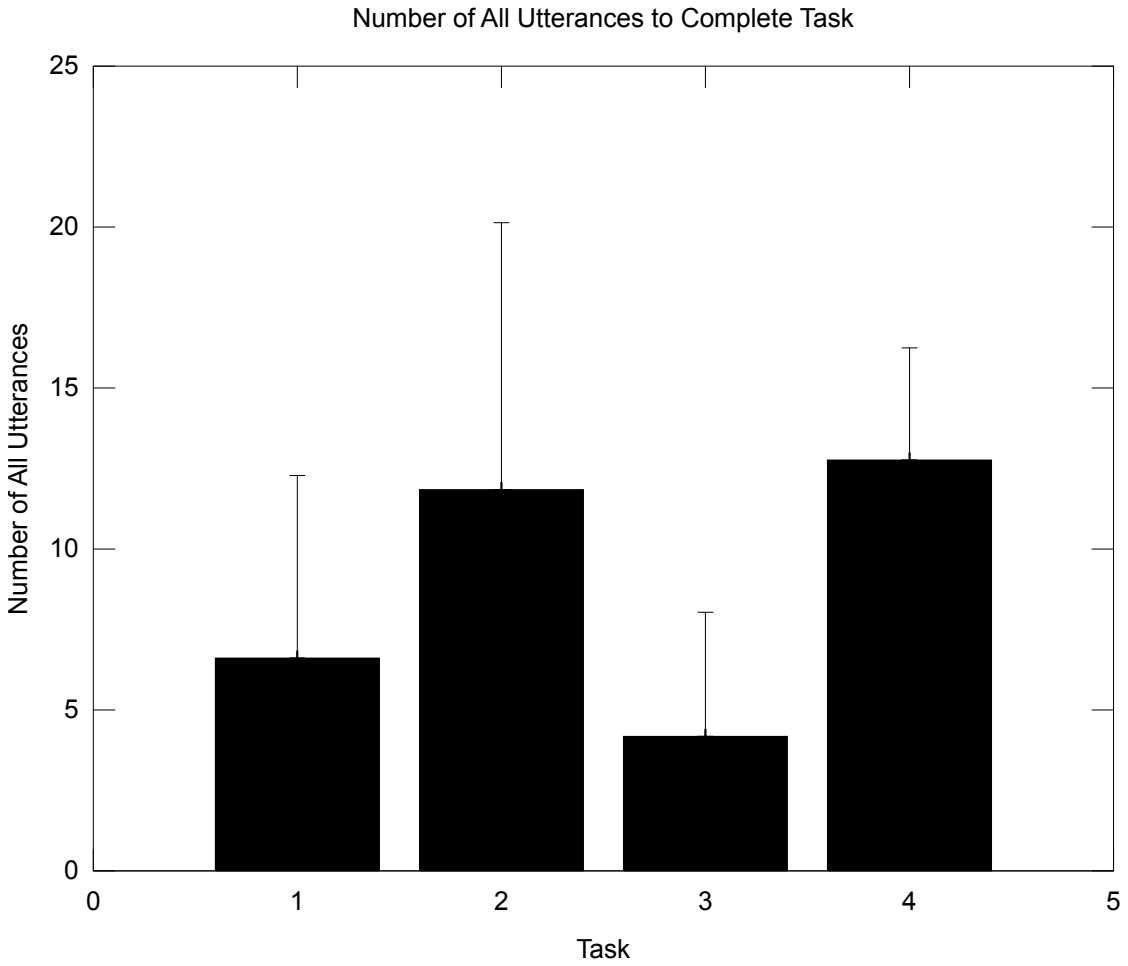



**Figure 6.1:** The number of utterances needed to complete each task is displayed above. The big black bars represent mean value across users and the tiny black bars indicate standard deviation. The simplest task, number three, takes around three utterances to complete while the hardest task, number four, takes around twelve utterances to complete. The standard deviation is roughly equal for all tasks, except number two.

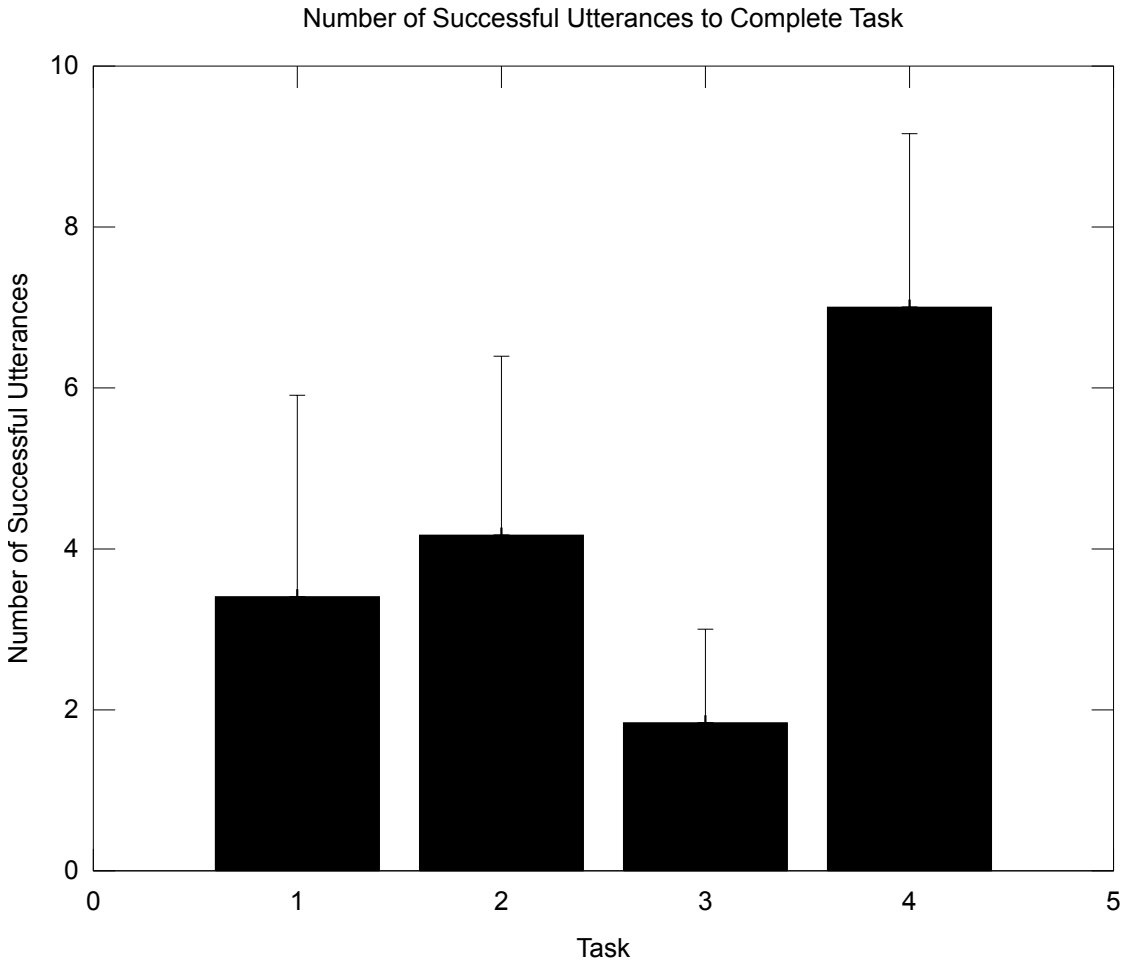



**Figure 6.2:** The number of successful utterances needed to complete each task is displayed above. Successful utterances in this context are those that contained a concept after the speech recognition step. The big black bars represent mean value across users and the tiny black bars indicate standard deviation.

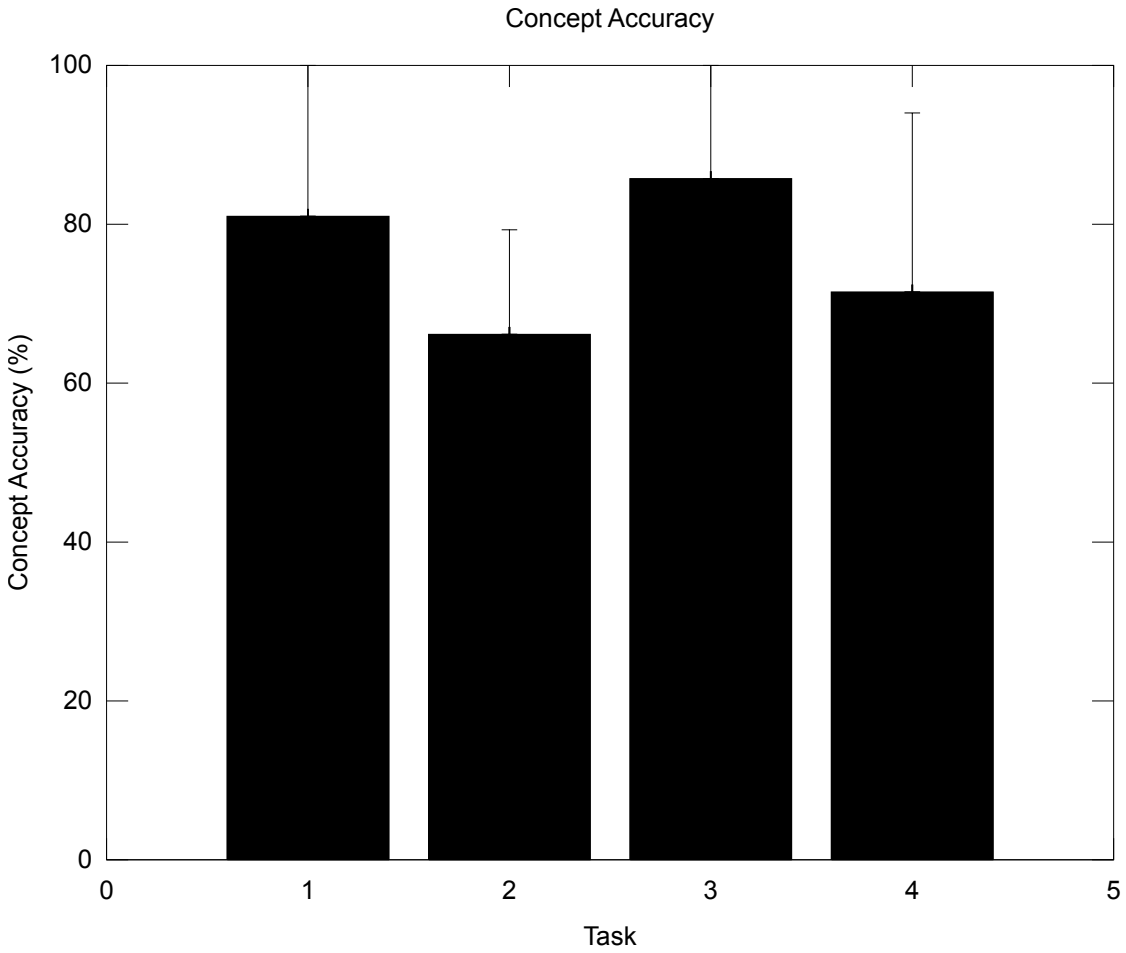



**Figure 6.3:** Concept accuracy for each task is displayed above. The big black bars represent mean value across users and the tiny black bars indicate standard deviation.

using the Nuance Vocon Hybrid ASR, a commercial and normally very accurate ASR. I suspect there was a problem with either the microphone, or the acoustic model used with the ASR. The microphone filtered noise, also reducing the audio output volume to the ASR. This could be a problem if the ASR wants input audio in a special way. Perhaps it wanted normalized audio where the maximum recorded amplitude matched the theoretical maximum. This is unknown, but if true it could certainly have been the cause of the problem. The second suspicion concerned the acoustic model. As described in section 2.1.1.1, the acoustic model models how words sound. If the acoustic model used in the ASR was made for a different language or a strongly differing accent, it may also have been the cause of the problem. My personal judgement holds it more likely that the microphone caused the problem though. The acoustic model is a standard model provided with the Vocon Hybrid ASR, and should be able to handle regular English speech.

```
U> I want to eat
S> Okay. Do something
S> Okay. Eat
S> Started navigation to The Cheesecake Factory
```

**Figure 6.4:** This figure exemplifies the problems detected with the nav_do plan during user testing. TDM gives feedback in two steps, first *do something* then *eat*, confusing the user. It is desired to skip the *do something* feedback and just use the *eat*.

### 6.2.2 Interviews

Several problems were identified during the test subject interviews. I will discuss each problem in a separate section, below.

#### 6.2.2.1 The nav_do Plan

The nav_do plan (see figure 5.7) results in undesired feedback because of its structure. It wraps verbs, such as `eat` and `refuel`, finding places for them. This means that when TDM tries to give feedback acknowledging this, it comes in two steps. Firstly, the action request is acknowledged – "do something". Secondly, the verb is acknowledged – "eat". Instead, it is desired that feedback comes in one step – "eat". See figure 6.4 for an example of the problem.

**Proposed Solution**

A potential solution is to rewrite the `nav_do` plan, constructing a new plan for each supported verb. For example, a `nav_eat` plan, a `nav_refuel` plan, et cetera.

Another solution would be if TDM enables better spoken feedback, or enables better control over spoken feedback.

#### 6.2.2.2 Competing with Apple SIRI, Google Voice Search

The test subjects had tried the top commercial competition. For example the Apple SIRI[1] or Google Voice Search[2,3] (GVS) before trying the dynamic demonstrator. They thought the performance and accuracy was significantly lower for the dynamic demonstrator compared to competition. The SIRI and GVS both have one thing in common, one thing that makes them different from the dynamic demonstrator. They run on powerful servers in the cloud.

The problem in the dynamic demonstrator is that TDM works natively with a local ASR using a grammar based language model. A cloud based, freeform, ASR (one which

[1]Apple SIRI website: `http://www.apple.com/ios/siri/` [Accessed Oct 26th, 2012]

[2]Google Mobile website, voice search: `http://www.google.com/mobile/voice-search/` [Accessed Oct 26th, 2012]

[3]Google website, showing the voice search feature: `http://www.google.com/insidesearch/features/voicesearch/index.html` [Accessed Oct 26th, 2012]

can natively recognize any utterance) could be used to allow more robust recognition. In this case, its output must be parsed in a robust way to accurately determine utterance concepts for TDM.

Unfortunately, such a parser does not exist for TDM today, but if it did it could well increase ASR accuracy to the same levels as for SIRI and GVS, given that the same ASR is used. TDM however requires data, such as places, to be declared on initialization, for example in the ontology resource (see section 3.1.4). If the solution with dynamic application reloads (section 5.3.1) that was introduced with the dynamic demonstrator is used, places can be loaded dynamically. Since the ASR is cloud based, there would be no performance problems caused by ASR compilation (see section 5.4.1). Many place names would still not be supported however, unless covered by the cloud based ASR.

#### 6.2.2.3 Performance

With dynamic data compilations comes the performance problem mentioned in section 5.4.1. The user tests revealed that test subjects become confused during compilation because there is no feedback and it takes too long time. They are not sure what is going on, so they assume that everything is normal and continue to speak. This results in an unexpected ketchup-effect behaviour once compilation finishes.

**Proposed Solutions**

One possible solution is mentioned in section 5.4.1.

Another solution, which is much easier to implement, is to improve control over spoken feedback with TDM. Doing this, at least the user becomes aware of what is going on. However, users will still compare with SIRI and GVS and be disappointed, so it is no long term solution.

#### 6.2.2.4 Grammar Coverage

When the test subjects' utterances were not recognized by TDM several times in a row they became confused. Either, they spoke an utterance not covered by grammar, or the place they used (if they used one) was not loaded (section 4.3.1). The user could not tell from the spoken feedback given by TDM.

**Proposed Solution**

Solutions were proposed by users. One was to display what was actually recognized by the ASR. This would work when the ASR returns a result which is rejected by TDM, but it would not work when no results are returned at all.

Another proposed solution was to display some supported utterances; utterances that would surely work. This is tricky however, since places are loaded dynamically and they are often too many to display all at once. This approach eliminates the concern that the spoken utterance may not be covered by grammar.

#### 6.2.2.5 Places and Context

If the user stops speaking, resuming after a while, the last mentioned place is still considered in context and will be used if a place can be implicitly referred in the dialogue. The problem is that it is kept in context no matter how long the pause is. Considering human-to-human dialogue, a place is normally not in context for longer than some seconds or minutes. However, considering the nature of human-to-human dialogue, a place could probably be referred naturally hours later as well. It is simply hard to tell without performing user tests or consulting literature on the subject.

**Proposed Solution**

It is possible to start a timer to remove the place in context after a certain amount of time. Another solution is to make this part of grounding (section 3.1.3), where TDM gives more pessimistic feedback the longer time it takes since the place was mentioned. Research must be performed to show what a proper solution is.

# 7

# Conclusions

The purpose of the thesis have been, as stated in section 1.1, to enable spoken dialogue (SD) control of an in-vehicle infotainment (IVI) system. It have also been to enable place names across the world in the dialogue.

## 7.1 Accomplishments

First, I constructed a prototype demonstrator using a static grammar, the static demonstrator. It provided full integration with the IVI system, but places were static and did not cover the entire world. I constructed the dynamic demonstrator next, addressing the static grammar problem to provide world wide coverage of places. Some shortcomings were identified and potential solutions to them were proposed.

I also conducted small scale user tests to assess the usability and performance of the dynamic demonstrator. They revealed shortcomings as well, which I have provided potential solutions for.

## 7.2 Significant Shortcomings

As mentioned in the introduction 1, a dialogue system needs to be highly natural, highly intuitive and easy-to-use, among other things. Several shortcomings break these requirements, but I consider the following two problems the most important as a first step.

During the user tests, speech recognition accuracy was low. Users compared the dynamic demonstrator to top competition today and were disappointed, both with performance and accuracy[1]. Accuracy plays a major role if the system is going to fulfill the requirements of highly natural, intuitive and easy-to-use.

[1]Speech recognition accuracy have improved significantly since the user tests were conducted. A new microphone setup and optimized configuration have resulted in accuracy levels very close to those of the top competition, as long as the user utterance matches the grammar.

The dynamic demonstrator also has a performance problem. I identified it in the dynamic demonstrator (section 5.4.1) and the user tests revealed it as well (section 6.2.2.3). To fulfill the requirements of highly natural, intuitive and easy-to-use, there must be no performance problem even when the number of places is heavily increased.

## 7.3 Future Work

The most important future work is to solve the performance problem. I have proposed two different solutions to it. One is to go the same path as the top competition, using a cloud based speech recognizer (section 6.2.2.2). However, this solution requires a robust parser in order to interpret the semantics of recognition results. The solution also restricts place names, since place names are only supported if they are covered in the language model of the speech recognizer. In addition, using the cloud is unsuitable in an automotive environment.

Another solution is to avoid compiling the entire speech recognizer grammar during run-time. This can be done with a speech recognizer that can handle dynamic grammars (section 5.4.1). If so, the static parts of the grammar (everything except the places) are compiled in advance while dynamic parts (the places) are linked during run-time. This solution keeps the dynamic demonstrator mainly intact while the performance problem is greatly reduced. It however requires support to be implemented in Talkamatic dialogue manager (TDM).

# References


Acapela (2012a). Acapela Website - How does it work? http://www.acapela-group.com/how-does-text-to-speech-work.html. [Accessed May 17th, 2012].

Acapela (2012b). Acapela Website - What is Text-to-Speech? http://www.acapela-group.com/what-is-text-to-speech-faq.html. [Accessed May 17th, 2012].

Boros, M., Eckert, W., Gallwitz, F., Gorz, G., Hanrieder, G., and Niemann, H. (1996). Towards understanding spontaneous speech: Word accuracy vs. concept accuracy. In *In Proceedings of the Fourth International Conference on Spoken Language Processing (ICSLP 96)*, volume 2, pages 1009–1012. IEEE.

Chen, F., Jonsson, I.-M., Villing, J., and Larsson, S. (2010). *Speech Technology - Theory and Applications*, chapter 11. Application of Speech Technology in Vehicles. Springer.

Fant, G. (2005). Speech analysis and features. In *Speech Acoustics and Phonetics*, volume 24 of *Text, Speech and Language Technology*, pages 143–197. Springer Netherlands.

Fung, D. (2011). CNET Australia Website - Volvo Sensus Review. http://www.cnet.com.au/volvo-sensus-339315932.htm. [Accessed Oct 10th, 2012].

Google (2012). Google Places API Website - Supported Place Types. https://developers.google.com/places/documentation/supported_types. [Accessed Oct 4th, 2012].

Howard, B. (2012). Extremetech Website - Cadillac CUE hands-on: How well does Caddy's tech stack up to Ford Sync and MyFord Touch? http://www.extremetech.com/extreme/130139-cadillac-cue-hands-on-how-well-does-caddys-tech-stack-up-to-ford-sync-and-myford-touch/2. [Accessed Oct 10th, 2012].

Kang, S., Lee, S., and Seo, J. (2009). Dialogue Strategies to Overcome Speech Recognition Errors in Form-Filling Dialogue. In *Computer Processing of Oriental Languages. Language Technology for the Knowledge-based Economy*, volume 5459, pages 282–289. Springer Berlin / Heidelberg.

Larsson, S. (2002). *Issue-Based Dialogue Management*. PhD thesis, University of Gothenburg. [Accessed Oct 2nd, 2012].

Larsson, S., Ljunglöf, P., Cooper, R., Engdahl, E., and Ericsson, S. (2000). Godis: an accommodating dialogue system. In *Proceedings of the 2000 ANLP/NAACL Workshop on Conversational systems - Volume 3*, ANLP/NAACL-ConvSyst '00, pages 7–10. Association for Computational Linguistics.

McTear, M. F. (2002). Spoken dialogue technology: Enabling the conversational user interface. *ACM Computer Surveys*, 34(1):90–169.

Nuance (2007). Nuance Recognizer 9.0 – Grammar Developer's Guide.

Sutton, S. and Cole, R. (1997). The CSLU toolkit: rapid prototyping of spoken language systems. In *Proceedings of the 10th annual ACM symposium on User interface software and technology*, UIST '97, pages 85–86. ACM.

The Norwegian Meteorological Institute (2012a). Vilkår for bruk av gratis data frå yr.no. `http://om.yr.no/verdata/vilkar/`. Written in Norwegian. [Accessed Oct 2nd, 2012].

The Norwegian Meteorological Institute (2012b). YR Web Services - Locationforecast - Documentation.
`http://api.yr.no/weatherapi/locationforecast/1.8/documentation`. [Accessed Oct 2nd, 2012].

VoiceXML Forum (2000). Voice eXtensible Markup Language (VoiceXML$^{TM}$) version 1.0.
`http://www.w3.org/TR/voicexml`. [Accessed Oct 19th, 2012].

Walker, M. A., Litman, D. J., Kamm, C. A., and Abella, A. (1997). PARADISE: A Framework for Evaluating Spoken Dialogue Agents. In *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics*, pages 271–280. Association for Computational Linguistics.

# A

# TDM Device Protocol

This appendix covers the protocol used for socket communication between the spoken dialogue adapter and the TDM application. When reading this, keep the TDM application as your point of view. If a message is sent somewhere, the sender is the TDM application. Likewise, if a message is returned from somewhere, it is returned to the TDM application.

The static demonstrator just uses two messages, RequestStartNavigation and RequestShowWeather. All the messages are used in the dynamic demonstrator though.

**RequestInitializeCompiler** Sent to the spoken dialogue adapter.

Tell it to initialize the place compiler (the one between steps 3a and 4a in figure 5.2).

```
<request_initialize_compiler
    absolute_config_path = "string value"
    project_name = "string value"
    subapplication_name = "string value"
/>
```

**RequestInitializeCompilerSuccess** Returned from the spoken dialogue adapter.

Is True if the place compiler was successfully initialized.

```
<request_initialize_compiler_success
    success = "True | False"
/>
```

**RequestStartNavigation** Sent to the spoken dialogue adapter.

Tell it to start navigation to the specified longitude and latitude in WGS84 format. If the destination is displayed somewhere it will use the specified main (e.g. name of the named entity) and secondary (e.g. address of the named entity) names.

```
<request_start_navigation
    longitude = "decimal.value"
    latitude = "decimal.value"
    main_name = "string value"
    secondary_name = "string value"
/>
```

**RequestStartNavigationSuccess** Returned from the spoken dialogue adapter.

Is True if the navigation was successfully started.

```
<request_start_navigation_success
    success = "True | False"
/>
```

**RequestShowWeather** Sent to the spoken dialogue adapter.

Tell it to show the weather at the specified longitude and latitude in WGS84 format in `day_offset` number of days from today. If a forecast for the target position is displayed somewhere it will use the specified main (e.g. name of the named entity) and secondary (e.g. address of the named entity) names.

```
<request_show_weather
    longitude = "decimal.value"
    latitude = "decimal.value"
    day_offset = "int value"
    main_name = "string value"
    secondary_name = "string value (default '')"
/>
```

**RequestShowWeatherSuccess** Returned from the spoken dialogue adapter.

Is True if a weather forecast was successfully fetched and propagated. Note that it can be True even if there were no listeners for the result on D-bus (see the weather API in appendix C).

```
<request_show_weather_success
    success = "True | False"
/>
```

**RequestCurrentPosition** Sent to the spoken dialogue adapter.
Request the current user position.

```
<request_current_position />
```

**RequestCurrentPositionResult** Returned from the spoken dialogue adapter.
Contains the longitude and latitude coordinates of the current user position. In WGS84 format.

```
<request_current_position_result
    longitude = "decimal.value"
    latitude = "decimal.value"
/>
```

**RequestDestination** Sent to the spoken dialogue adapter.
Request the current destination.

```
<request_destination />
```

**RequestDestinationResult** Returned from the spoken dialogue adapter.
Contains the longitude and latitude coordinates of the current destination. In WGS84 format. Unclear behaviour if no destination is set (check with RequestNavigationIsStarted first).

```
<request_destination_result
    longitude = "decimal.value"
    latitude = "decimal.value"
/>
```

**RequestNavigationIsStarted** Sent to the spoken dialogue adapter.

Request to find out if a navigation is currently active.

```
<request_navigation_is_started />
```

**RequestNavigationIsStartedResult** Returned from the spoken dialogue adapter.

Returns True if there is an ongoing navigation (it checks if a destination is set).

```
<request_navigation_is_started_result
    result = "True | False"
/>
```

**RequestCompilePlaces** Sent to the spoken dialogue adapter.

Request place compilation for places around the specified position in longitude and latitude coordinate. It is specified in WGS84 format.

If `global_places` is set to True, include important locations in a 1000km radius from the specified position.

If `local_places` is set to True, include important locations in a 2.5km radius from the specified position.

If `poi_type` corresponds to a supported named entity category, include important named entities of the specified category in a 2.5km radius from the specified position.

```
<request_compile_places
    longitude = "decimal.value"
    latitude = "decimal.value"
    global_places = "True | False"
    local_places = "True | False"
    poi_type = "string value"
/>
```

**RequestCompilePlacesSuccess** Returned from the spoken dialogue adapter.

Returns True if the compilation was successful.

```
<request_compile_places_success
    success = "True | False"
/>
```

# B

# Spoken Dialogue Adapter API

There are two different APIs of the spoken dialogue adapter. Firstly, of course for the adapter itself. Secondly, for the spoken dialogue resource in the Pelagicore resource framework (ResFW). Both are described below.

## B.1 The Service API

The adapter communicates with ResFW, and anyone else that is interested, over D-bus. Below are its specifications.

```
Session bus name: com.pelagicore.spokendialogue
Object path:     /com/pelagicore/spokendialogue
Interface:        com.pelagicore.spokendialogue.SpokenDialogue
Provided methods:
    adapter_set_ptt_held
        parameters:
            None
        signatures:
            in:  ""
            out: ""
    adapter_ptt_held
        parameters:
            None
        signatures:
            in:  ""
            out: "b" (returns the status of the push-to-talk
                      button, True if held, False if released)
```

```
named_entities_available
    parameters:
        named_entities_db_agent_object_path:
        (object path of the named entities database -
         spoken dialogue agent D-bus object)
    signatures:
        in:  "o"
        out: ""
named_entities_unavailable
    parameters:
        None
    signatures:
        in:  ""
        out: ""
weather_available
    parameters:
        weather_agent_object_path:
        (object path of the weather service -
         spoken dialogue agent D-bus object)
    signatures:
        in:  "o"
        out: ""
weather_unavailable
    parameters:
        None
    signatures:
        in:  ""
        out: ""
```

```
Provided signals:
    pending
        parameters:
            None
        signature: ""
    ready
        parameters:
            None
        signature: ""
    close
        parameters:
            None
        signature: ""
    ptt_status_changed
        parameters:
            None
        signature: ""
```

## B.2 The Resource API

The resource API uses a D-bus property, nothing more. It is managed by the resource superclass, `service.Object`.

```
Session bus name: com.pelagicore.resource.SpokenDialogue
Object path:     /com/pelagicore/resource/spokendialogueX
                 (where X is an integer instance identifier)
Interface:        com.pelagicore.resource.SpokenDialogue.SpokenDialogue
Provided methods & signals:
    None
Pelagicore D-bus properties:
    PttHeld (bool - True if push-to-talk held, False if released)
```

# C

# Weather API

The weather service provides two APIs. One for the service itself, the separate program, and one for the resource in the Pelagicore resource framework (ResFW). Both are described below.

## C.1 The Service API

The service communicates with ResFW, and anyone else that is interested, over D-bus.

```
Session bus name: com.pelagicore.weather
Object path:     /com/pelagicore/weather
Interface:        com.pelagicore.weather.Weather
Provided methods:
    requestWeather
        parameters:
            longitude: (Decimal WGS84 coordinate)
            latitude:  (Decimal WGS84 coordinate)
        signatures:
            in:  "ss"
            out: "s" (returns the HTTP response body
                      with the yr.no results)
    readyQuery
        parameters:
            None
        signatures:
            in: ""
            out: ""
Provided signals:
    ready
        parameters:
            None
        signature: ""
```

## C.2 The Resource API

The weather resource takes a forecast from yr.no and standardizes its format before forwarding it. It accepts requests with the `requestWeather` method and eventually emits the result with the `weatherResult` signal. Requests can come the regular way on the regular D-bus object, from the HMI; or through the spoken dialogue agent D-bus object, from the spoken dialogue adapter. These are described below along with the standardized weather data format output by the `weatherResult` signal.

### The Regular Weather Object

```
Session bus name: com.pelagicore.resource.Weather
Object path:     /com/pelagicore/resource/weatherX
                 (where X is an integer instance identifier)
Interface:        com.pelagicore.resource.Weather.Weather
Provided methods:
    requestWeather
        parameters:
            longitude:      (Decimal WGS84 coordinate)
            latitude:       (Decimal WGS84 coordinate)
            main_name:      (main name if displayed)
            secondary_name: (secondary name if displayed)
            day_offset:     (number of days from today)
        signatures:
            in:  "ssssi"
            out: ""
Provided signals:
    weatherResult
        parameters:
            main_name:      (main name to be displayed)
            secondary_name: (secondary name to be displayed)
            result:         (resulting array of dicts,
                             containing standardized weather data)
        signature: "ssaa{sv}"
```

### The Spoken Dialogue Agent Object

```
Session bus name: com.pelagicore.resource.Weather
Object path: /com/pelagicore/weather/spokendialogueagent
Interface:   com.pelagicore.resource.Weather.Weather.SpokenDialogueAgent
Provided methods:
    requestWeather (This method eventually emits the weatherResult
                    signal on the regular weather object, described
                    above.)
        parameters:
            longitude:      (Decimal WGS84 coordinate)
            latitude:       (Decimal WGS84 coordinate)
            main_name:      (main name if displayed)
            secondary_name: (secondary name if displayed)
            day_offset:     (number of days from today)
        signatures:
            in:  "ssssi"
            out: "b"     (success)
Provided signals:
    None
```

### C.2.1 Weather Data Format

This is the specification of the standardized weather data format.

```
standardized_result = list(
    several_elements_of: dict(
        "time":              Python datetime.time timestamp in format:
                             "%Y-%m-%d %H:%M:%S"
        "temperature":       Temperature in degrees: "decimal number"
        "temperatureUnit":   Temperature unit for degrees: "C | F"
        "windDirection":     Wind direction, north is 0:
                             "decimal number"
        "windDirectionUnit": Wind direction unit: "deg | rad"
        "windDirectionName": Wind direction name:
                             "N | NE | E | SE | S | SW | W | NW"
        "windSpeed":         Wind speed: "decimal number"
        "windSpeedUnit":     Wind speed unit: "m/s"
        "pressure":          Air pressure: "decimal number"
        "pressureUnit":      Air pressure unit: "hPa"
        "symbol":            Symbol id, yr.no-specific: "integer"
        "precipitation":     Precipitation during interval:
                             "decimal number"
                             (KNOWN BUG: Does not accumulate
                              full day precipitation)
        "precipitationUnit": Precipitation unit: "mm"
    )
)
```

# D

# Named Entities Database API

The named entities database provides two APIs. One for the database service itself, the separate program, and one for the resource in the Pelagicore resource framework (ResFW). Both are described below.

## D.1 The Service API

The database service communicates with ResFW, and anyone else that is interested, over D-bus.

```
Session bus name: com.pelagicore.namedentities

Object path:     /com/pelagicore/namedentities
Interface:        com.pelagicore.namedentities.NamedEntities
Provided methods:
    requestPois
        parameters:
            longitude:    (Decimal WGS84 coordinate)
            latitude:     (Decimal WGS84 coordinate)
            meter_radius: (Entities returned are
                           located within this radius of the
                           given longitude and latitude)
            types:        (List of Google Places types.
                           See details at the below address)
        signatures:
            in:  "ssias"
            out: "s" (returns the HTTP response body
                      with the Google Places results)

https://developers.google.com/places/documentation/supported_types

    requestLocations
        parameters:
            longitude:    (Decimal WGS84 coordinate)
            latitude:     (Decimal WGS84 coordinate)
            meter_radius: (Locations returned are
                           located within this radius of the
                           given longitude and latitude)
        signatures:
            in: "ssi"
            out: "s" (returns the HTTP response body
                      with the GeoNames results)
```

```
    readyQuery
        parameters:
            None
        signatures:
            in: ""
            out: ""

Provided signals:
    ready
        parameters:
            None
        signature: ""
```

## D.2 The Resource API

The named entities resource can send named entity category-specific requests to the database service, not requiring the resource user to have knowledge about the actual categories and their spelling. The resource also standardizes the results, providing the same format regardless of the actual data source. Requests can come the regular way on the regular D-bus object, from the HMI; or through the spoken dialogue agent D-bus object, from the spoken dialogue adapter. These are described below along with the standardized place data format output by the `result` signal and the named entity categories that were supported at the end of this thesis.

### The Regular Named Entities Object

```
Session bus name: com.pelagicore.resource.NamedEntities
Object path:     /com/pelagicore/resource/namedentitiesX
                 (where X is an integer instance identifier)
Interface:        com.pelagicore.resource.NamedEntities.NamedEntities
Provided methods:
    requestFoodPois       (places where you can eat
                           eventually emitted by resource)
        parameters:
            longitude:    (Decimal WGS84 coordinate)
            latitude:     (Decimal WGS84 coordinate)
            meter_radius: (Places returned are
                           located within this radius of the
                           given longitude and latitude)
        signatures:
            in:  "ssi"
            out: ""
    requestGasstationPois (places where you can refuel
                           eventually emitted by resource)
        parameters:
            longitude:    (Decimal WGS84 coordinate)
            latitude:     (Decimal WGS84 coordinate)
            meter_radius: (Places returned are
                           located within this radius of the
                           given longitude and latitude)
        signatures:
            in:  "ssi"
            out: ""
```

```
    requestPoisByType     (places of the specified category
                           eventually emitted by resource)
        parameters:
            longitude:    (Decimal WGS84 coordinate)
            latitude:     (Decimal WGS84 coordinate)
            poi_type:     (Named entity category)
            meter_radius: (Places returned are
                           located within this radius of the
                           given longitude and latitude)
        signatures:
            in:  "sssi"
            out: ""
    requestLocations      (geographical locations
                           eventually emitted by resource)
        parameters:
            longitude:    (Decimal WGS84 coordinate)
            latitude:     (Decimal WGS84 coordinate)
            meter_radius: (Places returned are
                           located within this radius of the
                           given longitude and latitude)
        signatures:
            in:  "ssi"
            out: ""
Provided signals:
    result
        parameters:
            result: (resulting array of dicts,
                     containing standardized place data)
        signature: "a{sv}"
```

**The Spoken Dialogue Agent Object**

```
Session bus name: com.pelagicore.resource.NamedEntities
Object path: /com/pelagicore/namedentities/spokendialogueagent
Interface:   com.pelagicore.resource.NamedEntities.
             NamedEntities.SpokenDialogueAgent
Provided methods:
    requestPoisByType     (places of the specified category
                           eventually emitted by agent)
        parameters:
            longitude:    (Decimal WGS84 coordinate)
            latitude:     (Decimal WGS84 coordinate)
            poi_type:     (Named entity category)
            meter_radius: (Places returned are
                           located within this radius of the
                           given longitude and latitude)
        signatures:
            in:  "sssi"
            out: ""
    requestLocations      (geographical locations
                           eventually emitted by agent)
        parameters:
            longitude:    (Decimal WGS84 coordinate)
            latitude:     (Decimal WGS84 coordinate)
            meter_radius: (Places returned are
                           located within this radius of the
                           given longitude and latitude)
        signatures:
            in:  "ssi"
            out: ""
Provided signals:
    result
        parameters:
            result: (resulting array of dicts,
                     containing standardized place data)
        signature: "a{sv}"
```

### D.2.1 Place Data Format

This is the specification of the standardized place data format.

```
standardized_result = dict(
    "type": "poi | location" (poi = named entity)
    "places": list(
        several_elements_of: dict(
            "name":      (required - the English name of the place)
            "longitude": (required - decimal WGS84 longitude)
            "latitude":  (required - decimal WGS84 latitude)
            "amenity":   (optional - the named entity category,
                                     only available for named entities)
            "address":   (optional - the address of a named entity,
                                     only available for named entities)
        )
    )
)
```

### D.2.2 Supported Named Entity Categories

Only one named entity category is supported by the named entities database – the `Food & Drink`. More are needed but in the dynamic demonstrator this is the only one. The reason is that the data request messages only support one category at a time (section 5.2.1.2).

# E

# Places

Below are two sections, outlining what was desired when initiating the thesis. The first section lists tasks relevant to the use cases of navigation and weather. Remember that two kinds of places are used, geographical locations and named entities. The second section covers named entity categories.

## E.1 Use Case Tasks

This section covers desired use case tasks for the navigation and weather use cases. No further explanation than the short headings are given. The idea is that tasks can be performed one-at-a-time or several in the same utterance, using spoken dialogue system.

1. Navigate to a location
2. Navigate to a named entity at or near a location
3. Navigate via locations
4. Navigate via named entities at or near locations
5. Display named entities at or near locations
6. Display weather at or near location or named entity
7. Display weather at specified time or within specified timespan
8. Change some settings
9. Change view mode, one of
   (a) Map tracking (Follows the user's vehicle on the map)
   (b) List of directions (Shows a list of navigation directions)

(c) Static map location (Center's the map on a specific location)

10. Change map view, one of
    (a) Egocentric view (Tracking)
    (b) Exocentric view
        i. Map view
            A. Track-up (Tracking)
            B. North-up (Tracking & Static)
        ii. Over-the-shoulder (Tracking)
        iii. God's-eye (Tracking)
        iv. Wingman's view (Tracking)
        v. Another person's view (Tracking)
11. Ask context-relevant questions, such as
    (a) Distance to destination
    (b) Phone number of named entity
    (c) Population of city
    (d) Country of location
12. Receive continuous instructions of navigation

## E.2 Named Entity Categories

This section declares some relevant and desired named entity categories[1].

1. Parking
2. Car repair service
3. Car wash
4. Gas station
5. Toll station
6. Ferry terminal
7. Airport
8. Railway station
9. Bus terminal
10. Bank
11. ATM
12. School
13. Emergency care
14. Entertainment
15. Food & Drink
16. Lodging
17. Recreation
18. Post office
19. Tourist information
20. Service Shop
21. Tourist Attraction

[1]Relevant and desired named entity categories have been derived from amenities used by OpenStreetMap. See `http://wiki.openstreetmap.org/wiki/Map_Features#Amenity` [Accessed November 17th, 2012].

# F

# System Evaluation Tasks

All test subjects were given the four tasks below during evaluation. All tasks were given in the same order.

1. Start navigation to a place where you can eat.
2. Start navigation to a city that has good weather the coming days.
3. Start navigation to a place where you can eat, in your destination city.
4. Show the weather at your current location.