Dynamic Updates on Points of Interest for Navigation Systems

Author: Bart Meulenbroeks

Supervisor: Dr. Jozef Hooman
Supervisor: Dr. Theo Schouten
Supervisor: Peter Goedegebure
Thesis number: 570

Acknowledgements

Writing this thesis was not an easy job and I am very greatful for all the help and support I have received over time from many people. Thanks to everybody who helped in a certain way and thanks for all your patience.

I also want to thank some people in particular. Firstly I want to thank Jozef Hooman for all the assistance he gave me during this time. His clear vision on how to write a thesis really helped me and he supported me in a very good way. I also want to thank him for the fact he kept on pushing me all the time.

I also want to thank Peter Goedegebure for all his time and help. I enjoyed the coffee breaks and all the technical discussions we had. These discussions made me think about certain problems in a different way which was very helpful. His clear view and reflection on all aspects helped me to come to the solution presented in this thesis.

Last but not least, I would like to thank Anouk Leppens, my lovely girlfriend, and my family. Although it took me quite some time to finish this thesis, they supported me all the time and I am very greatful for that support.

Contents

List of Figures

List of acronyms

ADAS	Advanced Driver Assistant Systems
ActMAP	Actual MAP
CCP	Current Car Position
EC	European Comity
CD	Compact Disc
DVD	Digital Video Disc
HTTP	HyperText Transfer Protocol
OEM	Original Equipment Manufacturer
POI	Point Of Interest
SVA	Siemens VDO Automtive
UMTS	Universal Mobile Telephone System
XML	eXtensible Markup Language

1 Introduction

Navigation systems are very common these days and it is nowadays possible to have one installed in almost every car. There are many different systems which all offer the same basic functionality, guide a driver from the Current Car Position (CCP) to a destination. The more cheaper systems just use an arrow to guide the driver to its location whereas the more enriched systems can use a large color screen to guide the driver. These larger screens make it possible to show a map (even 3-dimensional) on screen while it can also provide some graphical guidance at the same time.

Siemens VDO Automotive (SVA) is a developer and manufacturer of navigation systems. They develop systems which are installed in a car before it roles out of the factory or which can be build in afterwards (the after sales market). Lately, stand alone systems are also part of the set of products developed by SVA. There are systems developed for both the cheaper and the richer (multimedia) platforms.

Todays navigation systems offer however more functionality then just guidance from point A to point B. Some functionality is not even related to the navigation system like playing music or video on the system. An enrichment which is strongly related to a navigation system and which can comfort the driver are Points Of Interest (POIs). These are points of general interest like a fuel station, hotel or a shopping mall. Most navigation systems provide a database which also contains POIs. Adding POIs to a system makes it possible to add a variety of new functionalities. Some examples are

- choosing a POI as the destination (look for a hotel in the vicinity of your primary destination);
- alert points, warn a driver when they reach such a point (e.g., a speed trap);
- POIs on the map, show certain POIs in the surrounding of the CCP (e.g., gas stations).

More detailed information on POIs can be found in chapter 2.

SVA provides already functionality on POIs for a long time and this functionality has been growing over the years. SVA is one of the top players regarding POI related functionality and they still provide functionality which is not offered by other competitors.

Updating an in-vehicle database is nowadays done by inserting a CD or DVD into the system. This disc contains the newest version of the database and it is inserted or copied to the target system which then can use the latest version of the database. Getting this new version into the vehicle can take quite some time. Data suppliers have to provide the newest data and SVA has to convert the given data into a proprietary database. This database then has to be distributed so it can finally reach the navigation systems inside vehicles. This process takes some time and it is possible that some data is already outdated before it is even used by a vehicle. Because it takes some time before an update is inside the

vehicle the number of updates is also limited. Current databases are renewed once or twice a year.

This slow rate of updates was one of the reasons for the automobile industry to start a research project which had the goal to look into the possibilities to update an in-vehicle database more often. This project, called ActMAP (Actual MAP), was funded by the European Comity (EC) and its main goal was to create a framework that provides a way to get updates into a vehicle more frequently. The research project looked for a way to update the roadmap related databases inside the navigation system (specifically for Advanced Driver Assistant Systems (ADAS) which need an up to date map).

Until now, the concepts are not used by any navigation system and car manufactures are also somewhat cautious to use it because it has a great impact on the complete system. If an in-vehicle database is for example not correctly updated and it gets corrupted, then the in-vehicle database could no longer be used. This means that the driver cannot use its navigation system anymore. Some manufactures are however interested to start updating only the POI database because the impact is less.

The consortium did not specifically look at POI related databases although it was mentioned that the concepts could be used for them as well. This thesis describes the research which is done to find an answer on the question how to update a POI database inside a navigation system in a dynamic way.

1.1 Research question

As stated above, SVA looks for a way to update POIs in a dynamic way. Therefore the following research question is stated and will be answered in this thesis:

How to update POIs dynamically?

To find the answer to this main question four related sub-questions are formulated which help to find an answer on the main research question:

1. What are the requirements for dynamic updates on POIs?
2. Is the ActMAP framework suitable for dynamic updates of POIs? If not, what is a good alternative?
3. Given a certain framework, such as ActMAP or an alternative, which (design) choices have to be made to create a solution?
4. Is the proposed solution implementable and to which extend does it satisfy the requirements?

1.1.1 Approach

In order to answer the research question, we perform the following steps:

- The first step is to define the characteristics of POIs. In chapter 2 we give an overview in which we start with the description of today's POI databases and the POIs which are in it. After that, some new trends which will be important for the future are explained.
- Based on information and interviews, we give a prioritized list of requirements. This list makes clear which requirements are important for dynamic updates on POIs. Together with these requirements some use cases will be described to illustrate typical usage of POI updates. The results are an answer to question 1 and they are presented in chapter 3.
- To find out if the ActMAP framework is suitable, we study it in detail. A summary of the framework is given in chapter 4 to provide the needed knowledge on it.
- We evaluate the framework with respect to our aim. Chapter 5 starts with this evaluation. If the framework does not satisfies the needs, we describe an alternative. This will answer question 2.
- Based on the framework, in chapter 5 we discuss the possible solutions which fit in the framework. The final goal is to provide a complete solution which meets the requirements from chapter 3 and which answers question 3.
- In chapter 6 we answer question 4 by validating the proposed solution with respect to the use cases described in the second step. This is done by implementing the service centre and by simulating the behavior of a navigation system.
- We end this thesis with our conclusions which can be found in chapter 7.

2 Points Of Interest

Navigation systems already provide information on POIs for quite some time. It started with a few POIs, but nowadays the number of POIs in a database for Europe reaches the 1 million. Not only the number of POIs is increasing, also the information per POI has increased. This chapter explains more about POIs. The current situation on POIs is firstly explained in section 4.1 and section 4.2 is about the future situation of POIs.

2.1 Current situation

The first POIs on a navigation system provided the driver with some practical information like the name and location of a gas station, hotel or restaurant. This was some very basic information and every POI was a record which had some attributes. Over time the number of POIs increased and all sorts of POIs were added. In the beginning the information was provided by the map suppliers. After some time the so called third party data (TPD) suppliers also started to provide information on POIs. An example is ADAC which publishes a camping and caravanning guide every year. This guide was made available for navigation systems and provided the driver with a set of camping's in Germany and Austria. The map provider provides some very basic information on a POI like the name, address and location while the TPD suppliers started to provide more and more detailed information on every POI. It is even possible to add an HTML page to a POI which contains additional information (text and images) on the POI.

Today's databases have information on POIs from both map suppliers and TPD suppliers. It depends on the requirements of the OEM (car manufacturers like Renault, BMW and Kia) which information is exactly added to database. Based on these requirements, SVA puts the information from map and TPD suppliers into one database which contains all the POI related information. Putting this together is not trivial because all suppliers provide their data in different formats and hence data is first transformed into a single format. This data then undergoes several steps which finally results in a proprietary database which can be used by a navigation system from SVA. Every navigation system supplier (like SVA, Becker or Blaupunkt) has its own proprietary in-vehicle database format and everybody has to create its own database out of the supplied data.

Every OEM has its own requirements on which POIs are provided to an end user. In general an OEM chooses which products should be put into the database. Every product is delivered by a map or TPD supplier and it contains several categories which divide the POIs logically. A category can be divided into several sub-categories. The POI database in the example of figure 1 on the following page has several products and a part of product 1 (e.g., a hotel and restaurant guide) is displayed. This product has two categories, restaurants and hotels. The restaurants category is divided into several sub-categories like Chinese and Indian restaurants. These sub-categories and the category hotels contain the actual POIs. This makes it possible to create a tree of categories where each leave of the tree contains POIs.

A category can have 'rules' for the POIs which are in it. These rules tell something about the information which is provided for each POI and they make it

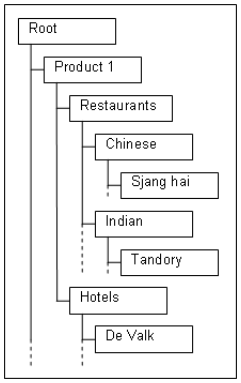

Figure 1: Category example

possible to add extra functionality. A rule for a hotel category can be that every POI contains information on the number of stars for this specific hotel. Another example is a price range which provides information on the price for a room for one night. Because this information is provided for all POIs in a category by a data supplier, it is possible to provide extra functionality like a searchable attribute. A driver can for example specifically search for a hotel with 3 stars in the vicinity. Another example could be the speed trap category. Every POI (speed trap) could provide information on the maximum allowed speed at that point. The navigation system can alert the driver when he or she reaches the speed trap and is driving too fast.

Next to the fact that a category can have 'rules' for the attributes on a POI, one can divide all the attributes of a POI into three groups. Some attributes are mandatory for each POI like the attribute(s) which define the location. Other attributes are category specific and there are also attributes which are optional (like the HTML info). Together the attributes provide all the necessary information which is needed for the navigation system to work for the driver.

The size of a database mainly depends on the requirements of the OEM. An OEM has to decide which products (with its categories and POIs) are added to the database. The number of products can vary between a few up to more then 100. Every POI in a product contains some data. The amount of data varies between 20 bytes to 1 kilobyte per POI. In total a database can be 2 megabytes for the small databases but the larger databases are up to 1 gigabyte in size.

2.2 Future situation

At this moment, POIs are stored as individual points in the in-vehicle database. This is an important reason to start with updates on POIs because they currently are isolated points which are simple to update compared to the road network (which has dependencies). Because there are no dependencies within the database it is possible for a POI to be more then once in the database. This holds, for instance, for the POIs which are for example registered in two cate-

gories. It is expected that more and more POIs will be in multiple categories so this could cause problems.

This is not the only relation which is expected in the future. One can also define other relations between POI which are already available in the database. An airport can be available as a POI but some systems also have separate POIs available which define a parking place or the departure hall from an airport. These POIs can be seen as a sort of children from the airport but are not stored as such. It could however be useful to do so, e.g., to ensure that removing the airport from the database implies that also all its children will be removed.

A POI can point to another POI on different levels. One can think of an attribute which points to another POI. The parent-child relation explained above is an example but one could also think of other relations which are not necessarily a tree. Another place which can contain relations to other POIs is the HTML page. An HTML page could contain links which refer to another POI or an HTML page of another POI.

Currently, a POI is stored in the in-vehicle database as a point which has a list of attributes. This is a very flat way of storing data and current development has created a tree instead a list of attributes. A tree makes it possible to store data in a more organized way. One could, for example, group all address related data instead of having several attributes which together form the address. As another example, consider the entry points which can be defined for every POI. An entry point is a point which hooks an entrance from a POI to the actual road network. A POI can however have multiple entry points and these points are currently flattened out to several 'numbered' attributes. Grouping this kind of data makes much more sense but is currently not yet supported by the system.

3 Use cases and requirements

Updating an in-vehicle database is possible in many ways. Based on information and interviews we first describe some use cases to give an impression of the updates which should be handled by this solution. Secondly the requirements are listed which should be fulfilled by the final solution.

3.1 Use cases

In this section, two use cases are described to give an idea what type of updates are expected. Every use case describes a situation. Within every situation there is some information required which could be updated. For each use case it is described what is important to update and what not. It is also indicated which data is already available in current systems.

3.1.1 Use case 1

In this first use case, a driver is using its navigation system. This system can be updated by a CD or a DVD. This disc contains (the information for) a new version of the POI database which can be copied to the hard disc of the navigation system.

The driver can use the installed POI database but the database gets outdated after some time. Therefore (among other things) the driver visits the garage where the car is quickly checked. During this short visit, a disc is inserted into the navigation system which contains the latest updates regarding the in-vehicle databases. The POI database is updated as well and this does not take too much time. The driver can leave the garage with a navigation system which is completely up to date. For the POIs, this means that the complete set of POIs is updated.

3.1.2 Use case 2

In the second use case, a family is driving from the Netherlands to the south of Spain to enjoy a long vacation. Because they have a caravan, they have decided to drive in two days to their destination. They have planned to stop somewhere in the south of France at the end of the first day to stay there for the night. It was not yet decided where to stop. This would depend on their progress in the first day.

Once they are driving in the south of France, they decide to find a nice camping to stay there for the night. They ask the navigation system to find a nice camping along the route. The camping should also have a swimming pool and have at least three stars. All this information can be provided by current systems but this information can be quite old and updated information could help them.

Current systems also provide the option to add a price range for a camping but this price range is static (an indication) while the prices for a camping place differ per period. If the system could update those prices, then the driver could

search for a camping within a certain price range. This dynamic information could be of use because it informs the family in a better way.

3.2 Requirements

In the previous section two use cases are given which give an impression of the problem. These use cases are based on information and interviews. In this section the functional and non-functional requirements are given which should be met by the final solution.

3.2.1 Functional requirements

This section provides a list of all the functional requirements. The requirements can be matched to both use cases described in the previous section.

- The old way of updating should be possible for a new system (from now on called an offline update).

  This means that a database can be delivered by a CD or DVD. These databases should be usable by the system which means that it can be used on its own. It should also be possible to renew the database by another CD or DVD.

- It should be possible to update POIs (from now on called an online update).

  There should be a way to update the POIs in a more dynamic way by using (for example) a wireless connection or a memory stick (or card).

3.2.2 Non-functional requirements

Below, the non-functional requirements are listed. They are ordered by their importance (the most important requirement is listed first).

- An in-vehicle database should be consistent.

  Updating an in-vehicle database should result in a database which contains better (i.e., more recent) information. If, for example, the wrong POI is updated in the database, it would have been better if the update was not executed at all. This is an example where consistency plays an important role because the database could get corrupted without the ability to repair it. Another example could be that a POI is removed while other POIs have a reference to it. It is questionable if such POIs should be updated immediately by the update supplier or by the internal software. The given solution should however describe for these problems how they can be solved.

- Updates should keep the data as up to date as possible.

  Updating the database should not result in a database state where some data is older then it was before the update.

- Use open standards as much as possible.

  A dynamical database is a new concept in the navigation branch which is not yet implemented anywhere but it is foreseen that this concept will be requested by customers in the near future. To make this concept a success for POIs, it is important to have support for the ideas by as many participants as possible. This is important because of the fact that it will only work if all participants cooperate. Note that there are many participants involved (multiple TPD suppliers, navigation system suppliers, OEM's). Therefore an open standard is preferred to make the acceptance easier.

- Support for multiple kinds of update facilities.

  Getting updates in the in-vehicle database can be achieved in many ways. It should be possible to get updates directly via a wireless connection but the in-vehicle database might be updated via a memory card or stick as well. This memory device could be used to transfer updates to the in-vehicle application using a computer the driver has access to.

- Limit the number of discs (CD or DVD).

  Car manufactures have to ship many discs to their garages in order to get the latest databases in every car. This is for some manufactures a pain in the eye so they will not accept a solution which requires even more discs (compared to the current situation).

- Updates by discs should be as fast as possible.

  There are situations where a driver which visits the garage for a quick check. The idea is that the driver can leave as quickly as possible. Therefore the time which is spend in the garage by the driver should be as short as possible.

- Available memory (inside car) has to be taken into account.

  A navigation system has limited resources so this should be taken into account. An update which is for example very large could simply not fit into the memory of the system so the system could not update the database.

- Capable to determine the in-vehicle database status.

  At this moment it is possible for the current databases to determine which database is in the system. This is beneficial for two reasons, for development it is important to know who is working on which version of a database and it is also important to help customers with their problems. Not knowing which version of the database is in the system, makes it harder to develop and to support a customer. Therefore, from new in-vehicle databases it should also be possible to determine their state.

- Extendibility.

  The solution should be designed such that it is easy to extend the system. This because of the fact that this feature is new and it is expected that it will be extended because a new feature is never complete in the beginning.

- Available bandwidth / costs to transfer data.

  One of the possibilities to get updates into the vehicle is a wireless connection. These connections currently have a limited bandwidth and there are costs for every transferred byte. Although the use of a wireless connection could be limited by using a memory card, it is still interesting to keep the size of an update small.

4 ActMAP

ActMAP presented a framework [3, 2, 4] to update an in-vehicle database in a dynamic way. This chapter summarizes this framework which is further discussed in chapter 5. The framework has two parts, the concepts and the exchange format. The concepts are used to set out a base and the exchange format uses the concepts and defines a way to implement it. Both parts are summarized in this chapter, starting with the concepts.

4.1 The concepts

This section looks into the concepts in the order they are introduced by ActMAP. The concepts are explained from a 'POI point of view' meaning that the examples which are given are based on POIs. All the concepts are described and possible ways of using the concepts for updating POIs are mentioned as well (which are not provided by ActMAP).

ActMAP has defined 8 concepts in total which together form the basics of the framework. These concepts are:

- Incremental updates
- Baseline map
- Layers
- Partitioning and version control
- Map consistency
- Filter criteria
- Permanent identifiers and location referencing
- Time model

All these concepts are described, one by one, in the remaining of this section.

4.1.1 Incremental updates

Databases inside a navigation system are quite large these days. A POI database can be up to 1 Gigabyte in size and its size is still growing. Updating the complete database at once is not an option because the size of a complete update could be in the order of (tens of) Megabytes. Today's newest wireless networks like UMTS provide a data transfer rate up to 2 Mbps but these speeds are only reached with fixed stations. The data transfer rate for stations which move at pedestrian speed are 384 Kbps and for a moving car 128 Kbps [1]. This means that it would take over 5 minutes to get an update of 5 MB in a moving vehicle ($\frac{5*1024*8}{128} = 320\,sec.$). It is also questionable if a driver is willing to wait 5 minutes on a certain (full) update which might be only for a very small part

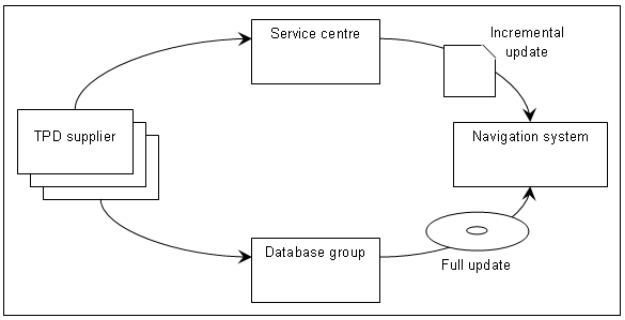

Figure 2: Service centre

of interest for the driver. Also the costs to transfer 5 MB into the car could be a bottleneck for drivers. The remaining part of the update could be about POIs which are for example located completely somewhere else or about POIs which are not of interest for the driver.

Not providing complete (full) updates on the complete database but smaller updates which only update a small part of the in-vehicle database is a logical decision based on the problems described above. This concept (partially updating the in-vehicle database) is therefore introduced by ActMAP as a solution. To reduce the size of an update even further they proposed the usage of incremental updates. This means that only the actual changes are send from the update supplier to the vehicle. These changes are based on the previous version of the database and they provide the database with new information which brings the database to a new state. The update chain is visualized in figure 2. The old path of providing a new version of the database is the bottom path (with its CD or DVD) while new incremental updates are send to a service centre. This service centre stores all the updates and it can send them to a navigation system when such a system requests particular updates.

Because the incremental update will be based on the previous state of the database, it is important that both the in-vehicle database and the database on the supplier side have the same state for the part which is updated. When a supplier provides a new update, it will calculate the differences between the old and the new state. The differences, also called the delta, are the actual update which is send to the service centre. The service centre will store all the deltas and it can send them to a navigation system.

Because a navigation system receives only deltas, it is important to start with the same state and to keep both sides the same. If this is not reached, it could cause the in-vehicle database to get corrupted and inconsistent which won't help the driver because wrong information could be provided. Therefore both sides need to start with the same starting point which is called a baseline map. This concept is described in the following section. The database is also split up into smaller parts which can be updated separately (by incremental updates). This is done by using layers (see section 4.1.3) and partitions (see section 4.1.4).

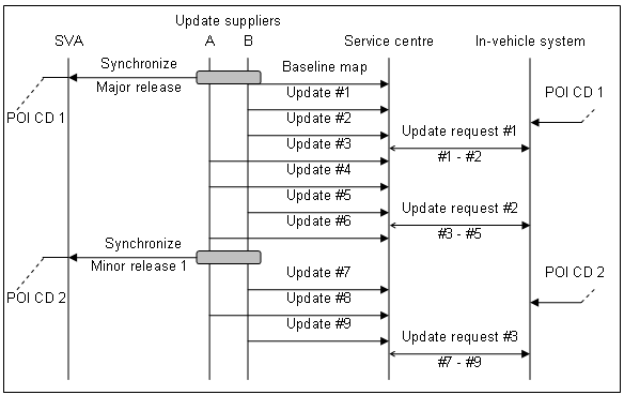

Figure 3: Major and minor updates

4.1.2 Baseline map

Incremental updates are based on the previous version of the database. Therefore the database needs a starting point related to which updates can be generated. This starting point should be used by both the updates suppliers and the navigation systems. It is called the baseline map and it synchronizes the update suppliers and the navigation systems in the beginning.

In the ActMAP framework a baseline map is provided by a map centre which produces the maps. In that case, there is one main data supplier which provides the baseline map. For a POI database the data can be delivered by several data suppliers and this data has to be merged into one baseline map.

If a new baseline map is provided, it is not only important to tell something about the actual data. The baseline map can also contain other data which is important for both update suppliers and navigation systems. One can think of the possible values a certain attribute of a POI can have. It is for example important that both sides agree on the number of stars a hotel can have. If an update supplier updates the data and says that the number of stars for a hotel will be 10 while a navigation system is not supporting this number, it could cause problems. Therefore it should be registered which data can be updated by an update supplier and how this data could look like.

Once a baseline map is generated, it is possible for the update suppliers to provide updates. These updates should be registered at the service centre which can distribute the updates to the navigation systems. An update will always be applicable to one baseline map. The service centre only has to register and store the updates and it does not necessarily need information on the baseline map.

This all is visualized in figure 3 where an example is given of the complete update chain (the time goes from top to bottom). There are two update supplier (A and B) which will provide updates. Before they can start to provide updates on a baseline map, it is necessary to synchronize them. At the first synchronization point (gray rectangle) the baseline map is registered at the service centre. The

service centre then knows that it can expect updates for this baseline map. At the same time, SVA (for example) receives the first release of the database (called the major release) which provides them all the information to create a CD. This CD contains a database which can be inserted into a vehicle and which provides the vehicle with an in-vehicle database. The vehicle can then contact the service centre to check for any updates. These updates can be send by all update suppliers to the service centre and will finally reach the in-vehicle database which then is updated.

The example also shows a second synchronization point which is not mentioned yet. After some time it could be decided to create a new CD which is based on the same baseline map. This version is called a minor release and it contains the database which is updated with all the updates which are registered at the service centre so far. Minor updates are introduced for two reasons:

- A system which gets the baseline map newly installed would have to get all the updates from the service centre if there would only be one major update. This could be an enormous amount of updates which have to be installed via the service centre.
- A system which already has an older version of the database installed can update just a part of the database via the service centre. It is also possible that certain parts of the database are not updated for some time. If a system wants to update these parts, this could also mean an enormous amount of updates which have to be installed.

In the example the second synchronization point is used to create a minor release of the baseline map. SVA can create a new CD which contains the updated version of the in-vehicle database. This database is installed in the vehicle and with the third update request from the in-vehicle system, just updates 7 till 9 are necessary. Update number 6 is already installed via the latest CD.

It is possible to update only a part of the database. ActMAP has introduced the layering concept and the partitioning concept to reach this goal. The next two paragraphs will describe both concepts in detail.

4.1.3 Layers

A rather late introduced concept within ActMAP is the layering concept. This concept makes it possible to divide the data into different layers which can be updated separately from each other. A layer should contain information which belongs logically together. In general (not POI related) this could mean that a database contains, for example, a layer for road elements and a layer for land use. For POIs one could use a single layer but it is also possible to split up all POIs into different layers.

There are two different types of layers defined by ActMAP, static layers and dynamic layers. A static layer is part of the database and contains the data which is defined so far. Data which is found on a CD (the major and minor releases) only comes from static layers. Dynamic layers can be defined in a baseline map but there is no actual data in such a layer when a major or minor release is

released. A dynamic layer can be used to add more dynamic information to a database. This information is ordered by events which means that it is possible to add, adjust or remove events from a dynamic layer. This feature is added to allow highly dynamic information and this information is placed on top of a static layer. An example of a dynamic layer could be a dynamic speed trap layer where temporary (non-fixed) speed traps can be added for some time. This dynamic layer can be put on top of a static speed trap layer which contains all the fixed speed traps.

ActMAP separates layers also in another way. They can be split up into standardized and user-defined layers. A standardized layer could be used by any update supplier to provide updates because the 'layout' of the layer is generally known. A user-specific layer has a layout which is not known by everybody and providing updates for a certain layer requires knowledge about this layer. For POIs there are no standardized layers at this moment.

The first way to divide the database into smaller parts is the use of layers. This is the 'horizontal' way to split data. The following paragraph describes partitions which divide the database on a 'vertical' way. It also gives an example on layers and partitions.

4.1.4 Partitioning and version control

Whereas layers provide a possibility to divide the data into logical groups, they do not solve the problem of getting updates about POIs far away from the current location. This problem is solved with the introduction of partitions. Partitions split a layer into smaller pieces and every partition can be updated separately. The size and shape of partitions can differ per layer and they can be chosen freely. A layer which contains for example only airports could have large partitions while a layer with restaurants could have smaller partitions.

Every partition will get an own version number. When one or more POIs in the partition are updated, the update supplier has to send one large update to the service centre to update the complete partition at once. The partition will then get a new version number. Because layers and partitions are used, a version number is assigned to a logical group of POIs within a certain area (a single partition) which reduces the number of version numbers.

The version number of each partition is captured in a timestamp which provides not only the version number but also information on the time the update was posted by the update supplier. This makes it possible to add extra functionality which is explained later on.

An example of the use of layers and partitions is given in figure 4 on the next page. It shows two layers (top) which each contain there own information (e.g., hotels and gas stations). The shapes of the partitions can differ as can be seen in the figure. The shapes of every partition are predefined and they can be included with the baseline map to provide this information to the navigation system.

All the information can be put into partitions but a problem rises when certain information crosses partition borders. This problem is discussed in the next paragraph.

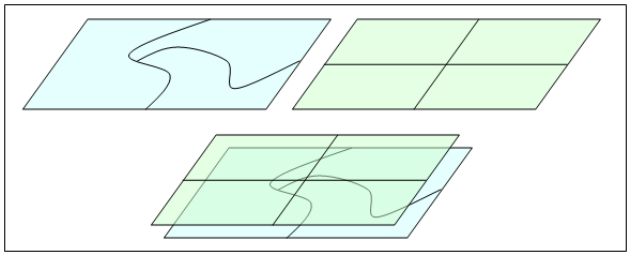

Figure 4: Layers explained

4.1.5 Map consistency

The introduction of partitions makes it easier to update smaller parts of the map but it also introduces a problem. There could be dependencies between different partitions. If only one partition is updated inside the vehicle while it depends on some data in another partition it is possible that the in-vehicle database gets inconsistent.

An example is given in figure 5. A possible POI could be an area (although not supported at this moment) like a nature reserve. This area is not located in one partition but in two partitions. If the POI is updated and the shape of the POI is changed, it has to be changed in both partitions. Therefore both partitions need to be updated inside the car which means that there is a dependency between both partitions. The result of only updating one partition is shown in figure 5 (right bottom picture). Only the left partition is updated properly while the right one remains its old version. As can be seen, the shape of the POI is not consistent anymore.

The dependencies problem is solved by using transactions. An update supplier which provides a new update knows if there are dependencies within the provided updates. It can put two (or more) dependent partitions in a single transaction and updated partitions which are within a transaction are always send together to a vehicle. The situation of figure 5 is then not possible anymore because both partitions are always completely updated in the same transaction.

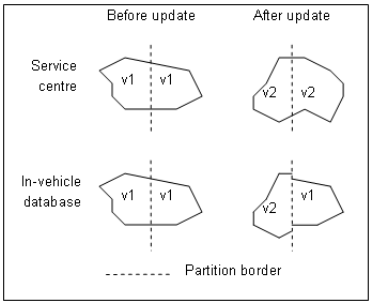

Figure 5: Map consistency

This solution is called clipping towards partitions.

If there are, however, many dependencies between partitions there could be a 'snowball effect' where every partition depends on some other partition(s). Then a small update can have a very large effect because of the dependencies.

ActMAP has proposed a solution to prevent this from happening. Instead of sending the complete update on a not requested partition, it is suggested to send only the affected information to the vehicle. The affected information is captured in a transaction which means that an update can have multiple transactions in it. This is called clipping towards transactions. It guarantees a consistent in-vehicle database but version control has to be extended. It is not enough to just store the version number of every partition, optional dependency related data could already be in the vehicle. Every transaction therefore gets a unique number so it can be registered which transactions are already send to the in-vehicle database. For version control this means that not only the version number of every partition needs to be registered but also the transactions (with their unique number) which are send to the vehicle. These transactions contain updates on a partition which is not yet updated.

The concepts which are mentioned so far describe the ideas to get updates in a vehicle in a managed way. The next paragraphs provide some information on other concepts which are needed as well.

4.1.6 Filter criteria

ActMAP has defined several filter criteria which provide a vehicle several ways to get updates from the service centre. These filter criteria make it possible for an in-vehicle application to filter the data for the needs of the application. All filter criteria which are defined by ActMAP are described below.

- Filtering per area

  A navigation system can specify an area which should be updated. This area can be defined by one or more partition ID's, a polygon, a region (e.g. administrative area) or a circle. If an area is updated by defining one or more partition ID's, the in-vehicle database needs the partitioning scheme. This scheme can be part of the baseline map. Then it is possible for the service centre to know nothing about the partitioning scheme. If polygons, regions or circles are used, then the service centre needs to know the partitioning scheme in order to determine which partitions are affected.

- Filtering per map content

  It is possible that an in-vehicle application is just interested in updates on certain data of a POI. An example could be an in-vehicle application which is started when the car is running low on fuel. The application will calculate the shortest route to the cheapest gas station. Therefore this application is interested in just the location and prices for fuel, a large description on the shop is not of interest for this application.

ActMAP has stated that filtering per map content can be reached by dividing the specific map content over different layers. Getting the data from a specific layer allows the user to get the right data. Another option defined by ActMAP is to filter the data for specific content (like certain attributes).

- Filtering per time

  It is possible for an in-vehicle application to get updates which are posted within a certain time span. A start and/or stop time can be given by the in-vehicle application to get certain updates which fall within this time frame. The version number is used for this filter criterion.

- Filtering per data source

  Every in-vehicle system should have the possibility to filter data on a specific data source. This means that it should be possible to define the map vendor and baseline map.

- Filtering per priority

  It is possible that there are multiple applications which use the same data but one application finds some data more important then another. If an update supplier provides lots of updates it is possible to give each update a priority number. The goal is to make it possible for an application to only get updates which have at least a certain priority level. Updates with a lower priority level could be skipped.

  A second idea regarding priorities is the idea to give an update supplier the possibility to provide updates with an extra high priority. These updates can for example be provided when an update supplier has provided an update which contains wrong information. The update supplier could then provide a new update which corrects the problem.

The filter criteria can be used separately but it is also possible to combine them in order to get a certain request. All these possibilities should provide a navigation system with a set of filters to get the right update into the vehicle.

In the previous sections, only partitions are addressed and it is not made clear how certain POIs in such a partition can be updated. The next section provides two ways to do this.

4.1.7 Permanent identifiers and location referencing

Updates which are produced by a data supplier are meant for one or more POIs. These POIs are located in a partition but within this partition a POI needs to be identified somehow. ActMAP has presented two options which are both discussed in this section:

- Permanent identifiers
- Location referencing

A first option is using permanent identifiers for every POI which means that every POI will get an ID. It is easy to change an attribute of a POI because one could provide an update which is for the POI with a certain ID. Some data suppliers already provide an ID for each POI when they deliver a new version of their data (full update) but there are also suppliers which do not provide any IDs for their POIs. It is vital that all data suppliers which provide updates for a baseline map have permanent IDs when this technique is used. These IDs should not only be permanent, they have to be unique as well. This means that it should not be possible to address more then one POI with an ID.

Location referencing is a second option which could be used to identify a POI. The idea is to identify a POI, which will be updated, by its location. There are several location referencing techniques like AGORA , ROSA or AGORA-C which provide a way to map certain characteristics from one map onto another one where both maps don't need to be the same. These techniques could be used for road elements but for POIs it is enough if a POI is referenced by its location. The disadvantage of location referencing is the fact that it is hard to give any guarantees that the right POI is addressed.

4.1.8 Time model

ActMAP introduces a time model which makes it possible to send updates which have a certain lifespan. It is possible to define a start time and/or end time for each update. Defining a start time means that a certain update is valid after the start time has passed. It provides the opportunity to get updates into a vehicle even before they are valid, so updates can be announced. The end time can be used to create temporary changes. These changes should not update an in-vehicle database after the end time has reached.

The time model could be used for critical situations where the provided information is time dependent. A simple example could be a new bridge for which a maximum height is defined for vehicles which use it. This maximum height could be used by truck specific navigation which checks for any low bridges on the route so they do not get stuck. It is important for trucks to have this information in time in the vehicle if they would solely depend on such information.

Therefore it is possible to get updates into the vehicle before they are even valid. Every update can get a lifetime for which it is valid. If it is known by the navigation system that such critical updates are provided one week in advance, then the system could contact the service centre once a week to get such critical updates onto the system. This could prevent the truck driver from getting stuck underneath a bridge.

4.2 Exchange formats

The second part defined by ActMAP is the exchange format. This format is an implementation of the concepts and it describes a way to get updates from the update suppliers into the navigation systems. ActMAP has decided to use XML to exchange data because XML is generic.

In total, four formats are defined which each have their own task. The four formats are shortly described in the remaining of this section.

- Update exchange format

  An update supplier provides updates to the service centre and the service centre provides updates to a navigation system. They are both using the update exchange format to get the updates from one place to another.

  The format specifies that the updates which are contained in one message are all about the same single layer in a baseline map. That is why the baseline map and the layer are part of the metadata which is included in the beginning of a message. After the metadata, the message consists of one or more update collections. An update collection provides an update on a certain partition. This update has an identifier for the partition and a version number (date and time). Within every update collection it is possible to have one or more transactions. A transaction can have dependencies with other partitions and it has one or more update operation. An update operation is performed on some items. An item can be for example a point, a line or an area and on every item, some data is changed.

- Update request format

  Getting information (e.g., updates) from the service centre is possible for a navigation system by using the update request format. This format specifies a way to send requests to the service centre. The service centre can answer a request with a corresponding response. For this response it can use the update request format as well or it can use the update exchange format if an update is requested. So the request format is always used to send a request from a navigation system to the service centre while the service centre can use both the exchange and request format for the response. This all is visualized in figure 6. The update exchange format is used to send updates to a navigation system and the update request format is used to answer all the other requests a navigation system can do.

  A navigation system can send several requests to the service centre and all requests are based on the filter criteria which are described in section 4.1.6. These requests can be divided into the next groups.

  – General information - these requests provide information on the support for the baseline map.
  – Updates - a navigation system can send different requests to the service centre in order to get updates.

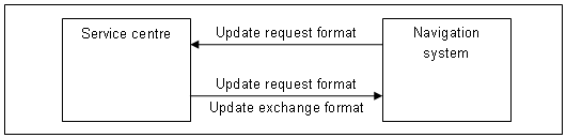

Figure 6: Different exchange formats

- Services - each navigation system can register services at the service centre. Requests for updates which are requested often can be stored in the service centre.

- Update notification format

  An update supplier cannot send updates to the service centre. It can only notify the service centre that new updates are available. The service centre should determine itself when it will get the new updates from the update supplier. An update supplier can notify the service centre by using the update notification format.

  This format simply provides some basic information on the updates the service centre can expect when it retrieves the updates. The notification contains the following information:

  - It contains information on the baseline map for which the updates are meant.
  - The update supplier can compress the updates and it can here provide the compression method which is used for it.
  - It is also possible to tell which partitions will be updated in the actual update.

- Data dictionary format

  The data dictionary format is used to define an agreement for the data which can be updated. A data dictionary is used for one or more layers and every layer has just one data dictionary. ActMAP summarized the data dictionary very short and it was not made clear how it should be used.

The 4 formats provide an implementation of the concepts from section 4.1 but it is not a complete solution because not all parts of the update chain are covered. In the next chapter, a design is presented which could be used to get POI updates in the vehicle.

5 Design

In the previous chapter the solution presented by ActMAP was summarized. This chapter first discusses this framework with respect to our requirements. In section 5.1 it is shortly evaluated if the conceptual framework could be used. If this is not the case, an alternative will be described. Based on the chosen framework, a conceptual design is presented in section 5.2 which presents a complete solution on a conceptual level. The exchange format, presented by ActMAP, is discussed in section 5.3 to see if the format could be of use while keeping the conceptual design in mind.

5.1 Using the ActMAP concepts

The concepts presented by ActMAP are summarized in 4.1 and in this section these concepts are discussed to see if they can meet our functional and non-functional requirements. We will see that the concepts can roughly meet our requirements and we will describe a total conceptual design in section 5.2.

The functional requirements require the possibilities to update the in-vehicle database by offline updates (updating by disc) and online updates (using a wireless connection or a memory stick).

For offline updates it must be possible to use just a CD or DVD to get a database on a navigation system. ActMAP has presented a baseline map with its major and minor releases to update the in-vehicle database without any other help. These concepts can be used for the offline update from the requirements.

For online updates, it is not possible to update the complete database at once (because of the size of an update). ActMAP has concluded this as well and they have introduced two concepts which make it possible to update the in-vehicle database just partly. They have introduced layers which make it possible to update a logical group of data. This concept could be mapped for POIs onto the categories which can be found in every POI product. These categories divide the POIs into logical groups and it makes sense to provide a way to update categories separately. The second concept, called partitioning, is the possibility to update only a small area instead of the complete map. A driver is normally only interested in just a small region and not the complete map. Therefore this concept could be used as well to provide the driver with information which is useful.

Hence, a first impression shows that the ActMAP concepts can be mapped onto the functional requirements. There are however many open issues which have not been discussed yet. In section 5.2 a complete conceptual design is presented which will show how the ActMAP concepts can be used. Also all the open issues will be addressed and solutions are provided for these issues.

5.2 Conceptual design

Using the ActMAP concepts is possible in many ways and ActMAP has left open many possibilities which have to be filled in. The remaining of this section

describes the open points and problems which are encountered. It results in a complete conceptual design which is suitable to update POIs.

For some problems it is possible to use multiple solutions while other problems only have one usable solution. It is also discussed why this solution(s) could be used best. Most problems and their solution(s) depend on other problems. Therefore it is sometimes necessary to make certain assumptions which are based on solutions for not yet discussed problems.

To make certain problems more clear, some algorithms are described in a sort of self describing pseudo code. This pseudo code looks like this.

```
some kind of algorithm;
```

The design starts with some general information on baseline maps. From that point on it is first discussed how offline updates can be implemented. After that the design choices for online updates are discussed.

5.2.1 Starting a new baseline map

A POI database has to be created before one can start updating it. Creating the database starts by defining a new baseline map. In this baseline map one has to make certain decisions in order to let the update chain work correctly. This section describes the decisions (list below) which have to be made when a new baseline map is created.

- A baseline map will have several layers which have to be defined. It is possible that multiple update suppliers provide updates on the same baseline map. In most cases it is expected that one update supplier provides the updates on a layer so it has to be registered who provides updates on which layers.
- Every layer will have a certain partitioning scheme and this has to be defined in advance as well. It is needed by the update supplier so it can group the updates together while the service centre and/or navigation system need it to know which partitions have to be updated.
- For each layer one must also determine the content of the data dictionary. This dictionary describes how the actual data could look like and it is needed by a navigation system.

Once these points are decided, it is possible to register the new baseline map at the service centre so the update suppliers can provide updates. The update suppliers can also provide the baseline map data to SVA (see figure 2 on page 12). SVA can then create the major release (a CD or DVD) out of the provided data. The disc can be inserted into a navigation system and the system should install the new database. Once it is installed it can be used by the driver.

After a database is installed, the system can contact the service centre for updates but one could also use the offline updates to update the in-vehicle database. The next section discusses the design choices for offline updates.

5.2.2 Offline updates

An installed baseline map can be updated by the use of minor releases. The data for these minor releases comes from the update suppliers and SVA creates a disc out of this data. This disc can update a system without the use of the service centre. A minor release provides a way to update the complete database up to a certain point in time (see figure 3 on page 13). ActMAP did not define how this minor release should look like. We discuss two possible ways this can be done.

1. Create a complete up to date database.

   Every time a new minor release is produced, the update suppliers provide the data which represents the current database to SVA. SVA creates a new database which is based on the data delivered by all the suppliers. This database is put on disc and is distributed to update the in-vehicle databases. The database can be updated by replacing the old version completely or by comparing both databases and update the old one.

2. Create a delta of the database.

   Instead of creating a complete new database, it is also possible to determine the differences compared to the previous minor or major release. These differences (the delta) can be given by the update suppliers to SVA or can be calculated by SVA (based on two versions of the database). SVA can put this delta on disc for distribution. An in-vehicle database could be updated by this delta.

If only a delta is put on disc, a navigation system might have a problem when an old version of a database (e.g. an old baseline map) is only available which is not supported by the delta. For these systems it is necessary that a complete database can be installed. Therefore a complete database (option A) has to be put on disc in order to update those systems. The additional information which is added to the major update, like the layering and partitioning scheme, has to be added to the minor releases as well for the system which have an old baseline map installed. By providing all this information, it is possible to get every in-vehicle database updated.

From this point of view it sounds logical to use this new release and simply replace the old version with the new one. This can be done by the following algorithm.

```
remove current database;
copy new database to navigation system;
```

Using the algorithm described above to update an in-vehicle database has some side affects which should be taken into account.

1. If a navigation system contacts the service centre to get updates (the online updates), then it is possible that the old database is already partly

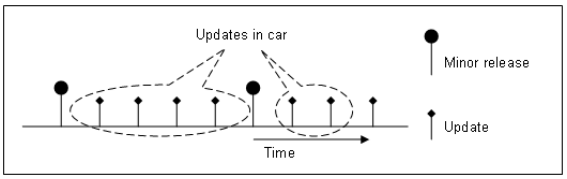

Figure 7: Update the database by a minor update

updated. It is even possible that the database in the vehicle contains updates which are newer then the data which is in the minor release.

This is explained by an example which is visualized in figure 7. Here some updates (from the service centre) and two minor releases are shown. Some of the updates are already send to the vehicle and the vehicle has updated the in-vehicle database with these updates. Once the second minor release arrives at the vehicle it is possible that already some newer updates are send to the vehicle (the two updates after the minor update in this example). These updates have updated the in-vehicle database with information which is not available in the minor release. Replacing the complete in-vehicle database by the minor release results in a database which contains partly older information compared to the prior version of the in-vehicle database. It is however a requirement to keep the database as up to date as possible and this requirement is not met when the database is replaced with older information.

2. Some systems have the possibility to store user defined POIs. These POIs could be placed in the same in-vehicle database. Simply replacing the in-vehicle database means that all user defined POIs are removed from the system. This side effect can however be solved by not placing the user defined POIs in the same in-vehicle database but in a separate one.

3. Copying a complete database from disc (CD or DVD) takes time which is not in line with the requirements which require a fast update from disc. If an old database is present at that moment then it is inevitable and a copy of the database has to be made. If an old version of the same database is available then the delta solution could be used as well.

Because of the first two side effects, it is preferable to compare both databases instead of replacing the old database. Also the first and third side effect show that replacing is not in line with the requirements. Therefore comparing could be a better option. Comparing both versions looks like this.

```
if ( current baseline map is equal to minor release
     baseline map ) then {
  for each ( layer in baseline map ) do {
    for each ( partition in layer ) do {
      if ( version of partition in current db is older
           then in minor release ) then {
        update partition to minor release;
      }
    }
  }
}
else {
  remove current database ;
  copy new database to navigation system ;
}
```

The comparing approach could be affected by the third side affect if the data is not stored efficiently on the disc. It would take too much time for the system to read the entire database from disc. Measurements show that the transfer rate to copy data from a CD onto a hard disk in a navigation system is around 2 Megabytes per second. So it would take over 8 minutes to copy a complete database of 1 Gigabyte. It would even take more time to compare both databases because more actions are required on the hard disc in order to update the in-vehicle database.

By storing the data on the disc in an efficient way (storing all version information together), it is possible to reduce the amount of data which is needed to be read from disc. If there are however partitions affected which need to be updated (probably most partitions will be), then all the data on the partition must be read from disc and which means that a lot of data must be transferred to the disc.

It is obvious that the in-vehicle database needs to be replaced when the current database and the minor release do not have the same baseline map. This also means that it is never possible to give always a solution for the requirement to get the update quickly into the vehicle. However, if the baseline maps are the same, then it is possible to use a delta (option B, discussed earlier on) to update the in-vehicle database.

If a complete copy is made of the minor release, then side effect 1 (loosing more recent updates) could also be solved by re-applying all the updates which are new. Therefore the navigation system should have to store all updates or it should get them again from the service centre. Storing the new updates requires extra storage room on the hard disc inside the navigation system while retrieving the updates from the service centre again requires more traffic between the service centre and the navigation system. This makes both solutions less interesting then using a delta. Therefore, the delta will be discussed next.

The advantage of a delta on disc is the fact that it will be smaller then a complete database because it only contains the changes. This means that it can be copied quicker on a hard disk inside a navigation system compared to a complete database. The last side effect (number 3) for a complete database

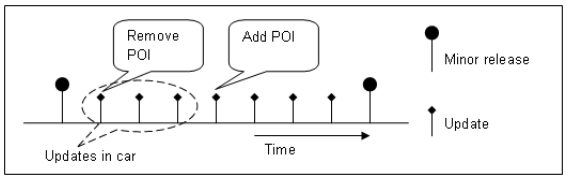

Figure 8: Updates stored in the service centre

is therefore solved for this solution. The update itself can be executed later on but a car could leave an update shop (garage) faster.

The first two side effects which were just mentioned above (when a database is replaced) are also not valid for the delta approach. The delta could be used in the same way as with the comparison method so no newer information is overwritten and no user defined POIs are removed.

A delta therefore seems to be a good alternative although this approach also has some point of attention which should be taken into account:

- The most simple way to provide a delta, is to provide a list which sums all the changes between both versions. This simple approach is sufficient for drivers who do not update their database using the service centre (online updates). For drivers who contact the service centre it could cause problems which are not obvious on first hand. This problem is explained by a simple example.

  A driver is updating its system by contacting the service centre. In figure 8 it is visualized that seven updates are stored in the service centre between two minor releases. The first 3 updates are already in the car and the first update removes a POI from the database. Some time later, the update supplier has decided to add the same POI again to restore a mistake. The update does not reach the car before the next minor release is installed.

Removing and adding the same POI within the database results in the same state for the database and this is not detected if we compare two successive minor release points in the database. So it is not part of the delta which arrives at the vehicle. This results in an inconsistent updated database where, in this case, the POI is no longer available.

This problem is not solved by comparing only the two points in time but one has to take all the updates into account. One has to check which data is changed (meaning added, changed or deleted) during this time period and all these changes should be mentioned in the delta. One could do something like this.

```
some kind of algorithm;
remove current database;
copy new database to navigation system;delta := empty();
for each ( layer in baseline map ) do {
  for each ( partition in layer ) do {
    if ( there are updates on this partition ) {
      partitionUpdate := empty();
      for each ( update of this partition ) do {
        add update to partitionUpdate;
        optimize partitionUpdate;
      }
      add partitionUpdate to delta;
    }
  }
}
```

The updates on every partition are gathered and put together. This complete update provides all the data which is of interest for the partition. It is possible to optimize the information partly. If for example the name of a POI is changed two times in a row, then the first change could be left out because it would not change the final state of the database. Doing these kinds of optimizations should be done carefully because it is easy to make a mistake (as mentioned in the beginning of this point of attention).

Every partition which is affected some times by one or more updates is covered this way and is added to the delta. The result is a large update which provides information per partition to get from the previous state of the database to the new state.

- For a delta one must pick a point in the past to calculate the differences between both versions of the database. The most logical point to pick is the previous major or minor release. This means however that every minor release can only be installed if the previous major or minor release is installed as well. Skipping one minor release means that all the remaining releases (the deltas) are not of any use and that the navigation system needs to use the complete database (which is always provided). Using the complete database is however not preferable because it will take more time to update the in-vehicle database.

  A single delta can solve this problem for the previous minor release but it is possible to provide multiple deltas to reduce the problem even further. This means that not only the previous release is supported by a delta update but also earlier releases. A CD will then contain several deltas (discussed below) and a full version of the database which can be used in cases where the deltas are not of any use.

  There are two possible ways to define multiple deltas, both are described below.

  - A successive list of deltas is provided on every disc which is distributed. A navigation system has to check which minor release was

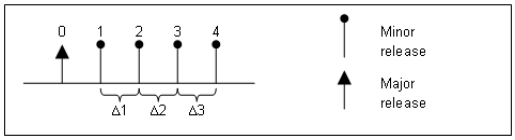

Figure 9: Delta storage 1

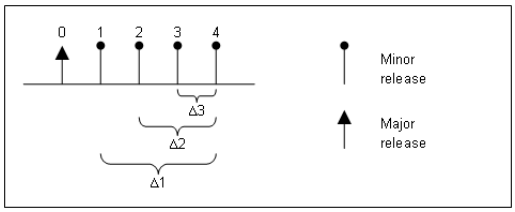

Figure 10: Delta storage 2

installed latest and all newer deltas should be installed on the system starting with the oldest one.

The example from figure 9 illustrates this. There is one major release (0) which is followed by 4 minor releases (1 - 4). Minor release 4 is the latest release and a disc is created for this version. Together with the complete database it is possible to add 3 deltas on this disc. Each delta is based on the changes between two prior minor updates. If the in-vehicle database has installed minor release 2 but did not install version 3, then first delta 2 and then delta 3 have to be installed in order to get the database updated up to minor release 4.

– Multiple deltas are calculated where every delta is based on another previous minor release. A navigation system now only has to install one delta, the one which is based on the latest installed minor release in the car.

  In this example (figure 10), there is again one major release and 4 minor releases. The disc providing minor release 4 again contains the complete database and it also contains three deltas. This time all deltas are based on the latest version of the database and a previous minor release state of the database. Only the delta based on the previous installed minor release has to be installed to update the in-vehicle database up to minor release 4.

Both solutions can be used and the deltas can be calculated as mentioned before. Not one of the solutions is specifically preferable. Assuming that all updates (between two minor releases) are of the same size, the first solution requires a linear disc space with respect to the number of deltas.

$$O(n) = n$$

The second solution, on the other hand, requires the triangular numbers (1, 3, 6, 10, 15, etc.) for disc space. This because of the fact that all the

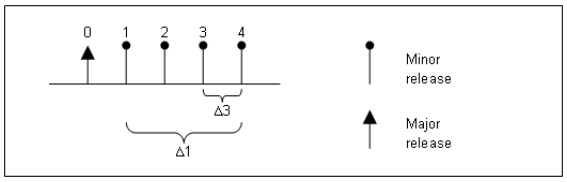

Figure 11: Delta storage 3

previous updates have to be added to a new delta as well. This results in a exponential grow of disc space with respect to the number of deltas.

$$O(n) = \frac{n \cdot (n+1)}{2} = n^2$$

Every delta is only once calculated when the first approach is chosen which also makes it somewhat easier to produce the minor releases. On the other hand, spending some more time and disc space results in deltas which are easier (= quicker) for a navigation system to install. Therefore it is concluded that both solutions can be used as a possible solution.

The complete database takes space on a disc and all deltas which are added contain redundant information. When the number of deltas increases, so does the amount of space required to store the deltas. There is a possibility to reduce the required disc space by not supporting all minor releases directly. As long as an earlier released minor (or the major) release is supported by a delta, the system could use this delta as an starting point to update the system from. The disadvantage of this technique is the fact the delta, which is then used, contains much information which is already in the vehicle. The in-vehicle database is updated again with this information. It is however a good fallback method to reduce the number of deltas.

This reduction is explained by an example in figure 11 which is comparable to the previous example in figure 10 on the previous page. In this case delta 2 is left out and this delta is not put on disc. A navigation system which has minor release 2 installed latest cannot use delta 3 to update to minor release 4. It can however use delta 1 although this delta will contain information which is already partly present in the system. The state of the system after updating it with delta 1 will however be correct.

The same approach can be used for the first delta approach where it is possible to take several deltas together to form a composite delta. This delta can then also be used for multiple minor releases in that particular case.

It is already reasoned before that there must always be a complete database available. Replacing or comparing the in-vehicle database is not an option if the database needs to be updated quickly. Because a delta is only a subset of the data which can be found in a complete database, a single delta will therefore always be smaller which makes it quicker to copy on a navigation

system. Therefore the requirement to get updates as quickly as possible on a navigation system is met. It is possible to place more then one delta on a disc in order to support more prior minor releases so this is preferable. The maximum number of deltas which are put on disc should be as many as possible to support a navigation system in the best way possible. The way the deltas are created is an open point which is not studied in this thesis.

With the use of major and minor releases it is possible to update an in-vehicle database without the help of the service centre (the offline update). The described methods also work for vehicles which do contact the service centre to get updates in their cars. In the next section it is discussed how products can be mapped nicely into the ActMAP concepts.

5.2.3 Splitting POI products

Before it is possible to create a database, it has to be decided how to split the data (POIs). ActMAP uses the layering concept to split the data such that it can be updated separately. This section shortly discusses the possibilities to use baseline maps or layers to split up the products inside a POI database.

A navigation system has one or more POI databases and every database contains one or more products. These products are completely separated and never have any relationship. An OEM decides which products are put together to form a database and it is possible that the products in a database change over time.

The POIs in every product can be split up into one or more layers. How this is done is discussed in the next section. For products there are two options which are discussed below:

- A baseline map defines which data there is in a database so from this point of view it makes sense to put all the products in the same baseline map. This results in a single baseline map for a POI database in a navigation system where all data is put in together. Every product has its own set of layers in the baseline map to split the products inside the baseline map.
- Because products are not related to each other, it is possible to create a baseline map for each product. Every baseline map has its own layers to store the POIs which are part of the product. This means that the in-vehicle database has to support multiple baseline maps.

The advantage of the last option is the fact that a product can change over time by creating a new baseline map. The other products can be left in place so the update chain is not disturbed for them. If the layering of a single product is changed in the first solution, the complete database needs a new baseline map which means that the other products need to be replaced as well.

It does not matter that much for the service centre which option is chosen and there are no conceptual changes for the previous discussed solutions if multiple baseline maps are used. A navigation system however will have to support multiple baseline maps for a single POI database if the second option is supported. There should also be a way to inform a navigation system to delete a certain

baseline map if it is replaced by another one but this is a relative simple action as well.

Separating products makes it easier for the update suppliers because they do not have to make a combined baseline map. From that point of view it is advisable to put every product into a separate baseline map.

Every product has a set of POIs which can be divided over one or more layers. The options to do this are discussed in the next section.

5.2.4 Splitting POIs in a product

A single POI product can have many different POIs which a driver does not want to update at once. Therefore POIs can be put into different layers which can be updated separately. This section discusses this topic and the problems which could be encountered.

In section 5.1 it was already discussed that it sounds logical to use layers to update for example separate categories in a product. This is a way to split the POIs such that they are divided over more layers. One could also put an entire product into one layer if a product is small. This solution is however not advisable for larger products which contain many POIs because it could take quite some time to update a small part (area) of such a product.

These POIs are normally divided over several categories which can have sub-categories (and so on). A logical solution to prevent a complete product to be updated at once is the introduction of multiple layers in a product. Every layer could cover one or more (sub-) categories. The categories divide the POIs in logical groups so it makes sense to use them to split a product over multiple layers. An example is given in figure 12 on the following page where a product with some of its categories is displayed. The product has a category religion which has some sub-categories. There is among other things a sub-category containing all chapels and a sub-category containing all churches. Both sub-categories have an own layer which makes it possible to update only chapels or churches. The category monuments can have multiple sub-categories but they are all put into one layer because the number of monuments is relatively small. Dividing all categories like this should finally result in a list of layers which covers all the (sub-) categories (and so the POIs) in a product.

This solution divides the POIs nicely into logical groups but there still is a problem. It is possible for a POI to be part of multiple categories. An example is a church which, of course, can be found in the category churches. The same church can however also be a monument so the same POI can be found under the category monuments as well. This problem could be solved by joining both layers but for some products this could end up with one single layer which was not intended (the layering was introduced to split the data). We describe three possible ways to solve this problem.

- Store a POI multiple times in the database, once per layer.

  This solution is applied in the current database. The POI is simply copied into all the categories where it can be found. The advantage of this solution is the complete separation of the layers so there are no dependencies

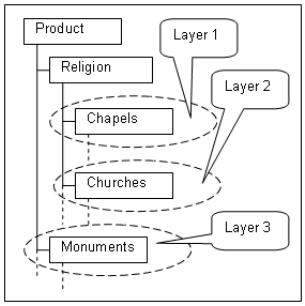

Figure 12: Layering example

at all. However, it has the consequence that a database contains some information multiple times. The church from the example has (for example) the same location, name and address in the church-category as it has in the monument-category. The fact that the information is stored more then once makes it possible that a driver gets different information about the same POI from different categories. A category could be updated while the other is not. This side effect might not be acceptable but the other two options do not have this problem.

- Store a POI in a cross-category layer.

  Besides the layers for every category, one could define one or more layers to cover the POIs which are in more then one category (also called cross-category). It is possible to define just one cross-category layer. This layer can be used to store POIs which are part of multiple categories. If any layer in the product is updated, this layer has to be updated as well because it could contain POIs which are part of that layer. This approach also has a drawback. When many layers share the same single cross-category layer, all these layers have to update this layer as well. The change that there are however POIs in the cross-category layer which are of interest decreases by the number of layers which use the cross-category layer.

  This problem can simply be solved by introducing more layers for cross-category POIs. There are however many ways to define these layers but the number of layers can however grow very quickly. We use the example of figure 12 to make this clear. As mentioned before, it is possible for a church to be a monument as well so an extra layer can be added to store all churches which are also monuments. A chapel can be a monument as well so we can do the same thing for a chapel, introducing a new layer. There might be situations where a chapel is build inside a church which again introduces a layer. To finish it up, there are such POIs which are a monument as well. The result is large list of cross-category layers which can end up as a mess quite easily. Even with this small number of categories (three), it is already possible to define more cross-category layers and this number only increases. If we assume that every possible combination of categories gets its own layer, then the maximum number of layers is equal to

$$Max(n) = 2^n - 1$$

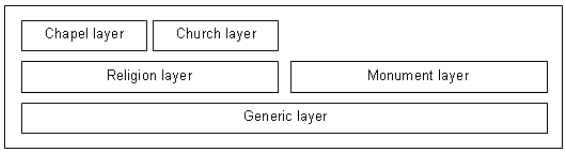

Figure 13: Hierarchical layers

where n is the number of categories. The number of all possible subsets of the set of layers is equal to 2^n. The empty subset is however not of intereset in our case (hence the minus one). Of course it is not always necessary to have all cross-category layers in the baseline map but the problem grows when POIs cover more categories.

There is another disadvantage when this technique is used. If a POI is added or removed from a category, then it is possible that the POI must be removed from one layer and added to another one. This means that all the data on a POI has to be send by the service centre to the navigation system. If, for example, the church is removed from the monument category then the complete data about the church has to be send to the navigation system while this same data is removed from the church-monument layer.

It can be concluded that the solution can be used but it has some major drawbacks which make it not preferable. The following solution does not have these drawbacks.

- Store the POI information in multiple layers.

  This approach does not keep the POI information in one layer (which is done by the previous solutions) but it separates the information on a POI over different layers. Every category can provide certain information on a POI and some information is generic for more categories and other information is category specific. The information which is generic should be stored in a different layer which makes it possible to share this information between categories. The whole idea is further explained by an example.

  We take again the example from figure 12 on the preceding page and we define some new layers (see figure 13). Every POI in this product has some basic information like the position, a name and an address. This information could be stored in a generic layer and every POI in this product is represented in this layer. The monuments category has (in this case) no sub-categories and every monument has some specific information like the year it was build and some educational text. This information is stored in the monument layer. The religion category has two sub-categories, chapels and churches. Because there is some generic information on religion related POIs, an extra layer is introduced which stores this data (like the type of religion). The specific information on chapels and churches is stored in separate layers as well. The result is a hierarchical list of layers which is visualized in figure 13.

  Storing thStoring the POI data in such a way makes it possible to store every POI only once in the database. Based on the information which is

available for a POI, it is possible to determine in which categories a POI is placed. If a POI is for example not mentioned in the church layer, then it is not a church.

The maximum number of (cross-category) layers is reduced by this approach because the layers have to be set up hierarchically. It can be calculated by

$$Max(n) = 2n - 1$$

where n is again the number of categories. This reduction can be of interest for large products where many POIs are spread over multiple categories. The only requirement for this approach is that layers have to be hierarchical.

The first solution can be used but an in-vehicle database will have to store POIs multiple times. The third solution is better then the second solution and it also prevents POIs from being stored more then once within the navigation system. Therefore this solution is preferred based on the information given so far. There are however also some side effects for this approach and they are discussed below.

- It is necessary to update multiple layers if a single category is updated. This is a small side effect which has to be taken into account. This would however have been a problem for solution two as well.
- The data of a single POI is stored within multiple layers. This means that multiple layers have to be updated in order to have POIs updated correctly. If not all layers are updated then one could get an inconsistent database. It is for example possible to update the monument layer from the earlier mentioned example without updating the generic layer. If a new POI is added to the monument layer, then the system needs the information from the generic layer as well. There are two ways which can be used to handle this problem:
  - The navigation system software has to be this smart that it updates all the relevant layers. This means that it not only has to update the layer which is directly attached to the category, but it also has to update the other underlaying layers.
  - There could be a dependency (transaction) between the updated partitions which means that it is not possible to update one layer without updating the others as well if this is needed. It is the solution which is provided by ActMAP to prevent inconsistencies.

  Both solutions have the same result and solve the problem but every solution has its own drawbacks. In the first solution, the navigation system is responsible to get the right data inside the system. It therefore has to send multiple requests to the service centre before it can update the database with the required information. Not having all the information inside the system makes the rest of the data worthless and it could even make the database inconsistent if the database is partly updated.

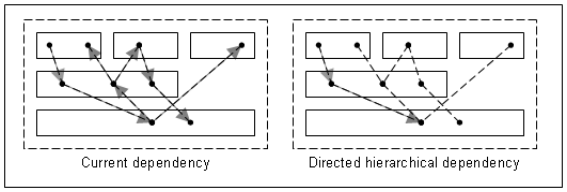

Figure 14: Dependencies explained

The second solution is somewhat cleaner because the data which is send to the vehicle is always consistent (by concept) and forms a complete set which can be used immediately in the system. From this point of view it makes more sense to use this solution. The big disadvantage of this approach is the fact that it could cause the 'snowball effect'.

The "snowball effect" is caused by the dependencies which link certain updates (partitions) together. Putting more dependencies within the same update makes it necessary to send updates on the affected partitions as well to a navigation system as well. This is visualized on the left side in figure 14 where several updates from different layers are shown. The hierarchical layers are again visualized as rectangles. The bottom rectangle is a generic layer and the rectangle on top of it is a layer specifying data for the two layers above it. The three top layers are related to a certain category. A navigation system sends a request to update a partition in the left top layer (each dot in the picture is a "request"). The arrows show which updates on other partitions in layers are needed in order to update that certain partition. Instead of only updating 3 partitions, in this case, 6 other partitions are updated as well. This example is still small but such an update could cascade through a lot of partitions which are affected and which have to be updated.

This problem can be solved by the introduction of directed hierarchical dependencies, meaning that every update in a layer only can have references to updates in a more generic layer. The result is shown on the right side in figure 14. In this case there can be only dependencies from the top to the bottom which results in just the 3 expected updates.

A directed dependency is not supported by ActMAP. This is not completely in line with the requirements which state that open standards should be used as much as possible. The "snowball effect" can however be prevented by introducing this solution. Therefore it is advised to use this solution. One could even try to get this feature embedded in the ActMAP concepts.

- Different layers can each have their own partitioning scheme. Updating a (category) specific layer can however mean that other, more generic, layers have to be updated as well. This is visualized in figure 15 on the next page where the generic layer has small square partitions while the specific layer has a single large square partitions (just to make things clear). If the large partition needs to be updated, then several partitions from the generic layer could be updated as well. In figure 15 on the following page all the

partitions which could contain updates on the specific large partition are shadowed as well (dashed).

If the navigation system would have to update the relevant partitions, then it would have to determine which partitions from the generic layer are of interest. But at the end of the previous bullet point, it was advised to use directed dependencies. These dependencies are set by the update supplier and the supplier can simply determine which partitions of the generic layer have to be updated as well when the specific layer is updated. If this solution (directed dependencies) is used, then the partitioning schemes of the different layers do not matter. This again makes the directed dependencies preferable.

- The generic layers can contain information on many POIs. Because of that, it is possible that a small request results in a big update because other information from other POIs is updated. This is a side affect which is inevitable. However, it can be limited by making the generic partitions very small. This is also done in figure 15. If only one POI is updated in the specific partition (large one), then there is at most one generic partition which should be updated as well (it could also be possible that the generic information does not change). Making the generic partitions smaller reduces this problem but one should be aware that the size of the administration inside the navigation system grows as the number of partitions grows.

- If the information on a POI is stored in different layers, then one should realize that the information is not always updated synchronically. The information of a POI in a generic layer could be updated when none of the specific layers of the POI are updated. Looking at figure 12 on page 33 (churches, chapels and monument example), it is possible that the information on a church POI in the generic layer is updated while the church specific information stays the same. The information should be divided between the layers such that every layer can be updated without bringing the complete POI information in an inconsistent state.

  An example is given to make this more clear, we use the same example again. A monument has a monument specific item which provides some textual information on the monument. This text is not a plain text but it can contain certain keywords which have to be replaced before the text is visualized to the driver. One of the keywords makes it possible to add the telephone number of the monument to the text. The telephone

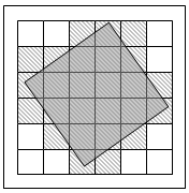

Figure 15: Partition updating

number is another item of the monument POI but this item is stored in the generic layer because every POI could have a telephone number. The telephone number is nicely inserted every time the driver sees this text. If the telephone number is however an optional item, then another update (not related to the monument part of this POI) could remove the telephone item from the POI. This way the textual information could not replace the keyword as intended which might be unexpected and unwanted behaviour.

- The largest side affect is the fact that the information on the generic layers can be retrieved multiple times. If the navigation system sends a request to the service centre, it has to tell the service centre which version it has on every partition. If the directed dependencies are used, then the navigation system will typically only provide information on the requested partition. This partition is however part of a specific layer while the information of the POI is stored on multiple layers. The more generic layers are not required for the request because the dependency makes sure that the required updates will arrive in the system.

One can use layers to split the POIs in a product. Having POIs in multiple categories is solved by storing the POIs multiple times in the database or by the hierarchical layering structure. The first solution is not that complex but it introduces redundant information in the in-vehicle database. The second solution is somewhat more complex but the redundant information is removed.

The presented solutions make it possible to divide the POIs over layers. Every layer is also split up into smaller parts, the partitions. The next section discusses the partitioning of layers.

5.2.5 Partitioning

Until now all POIs are nicely divided over several layers where each layer has its own partitions. The look of a partition is not yet discussed although they are used all the time. This section discusses the way a layer can be partitioned.

There are multiple ways to create a partitioning for a layer. There are a few points which should be taken into account when a partitioning is chosen.

- The number of POIs in a partition should be good. Having little POIs in every partition makes the idea of partitions useless, one could keep a version number for each POI as well. When there are too many POIs in a partition, then the updates can be too large or a driver gets information on so many POIs that not all of these POIs can be of interest for the driver. Therefore it is necessary to adjust the size of the partitions to the content of the layer. The partitions in an airport layer can for example be very large because the number of airports is limited. On the other hand, the partitions in a restaurant layer cannot have the same size as for the airport layer because there would be too many POIs in a partition.

- It is possible to update partitions separately but therefore it must be possible to determine which partitions must be updated. Once it is decided

that a layer is going to be updated, it must be decided as well which area (partitions) in the layer will be updated. Therefore the system must be able to determine which partitions map onto the area which is needed (or requested) to be updated. All the partitions which intersect with this area should be updated.

The area which should be covered has not always the same shape. There are two common situations which should be supported.

- Circles - these are of interest when a driver wants to have updates around the car.
- Corridor - this is of use when a route is calculated. The corridor is an area around the route which can be of interest for the driver.

The corridor can be dimensioned as a polygon which can be shaped quite strange. A very simple example is given in figure 16 to show how this could look like. The route the driver has to follow is covered by this polygon and so this area is updated. For a circle, one could use a regular polygon as well. This way all the areas which need to be covered are polygons.

These points are used to make a good decision on which partitioning to use. This decision is discussed in the remaining of this section. There are two main ways in which we can define partitions.

- Administrative areas
- Geometric shapes

Administrative areas (like countries, states or communities) could be used to split up a layer into smaller pieces. One could use for example countries in the airport layer. All the airports of the same country can then be placed in the same partition. There are however some drawbacks when this approach is used.

- Sometimes the administrative borders might not be the best place to split an area up.

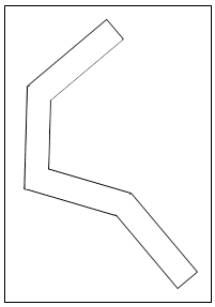

Figure 16: Route example

Although country borders are perfectly usable for an airport layer, they are not for a hotel layer. This layer could use the borders of cities for example but the number of POIs in a city can still be very large. Even a district in a city can have many POIs in it so the partitions could not get small enough.

- There is a dependency with map related data.

  An administrative area depends on other map info which is currently not available in the POI database. It should ask the maps database for this information. This is possible but it introduces a dependency which is not necessarily needed when geometric shapes are used.

Because of these two reasons it is decided not to use any administrative areas but only geometric shapes. If desired, one could still somehow use the administrative areas outside the navigation system when the geometric shapes are created. Then there is no dependency inside the navigation system between the POI database and the map database and the POI database can simply use shapes. One can distinguish two types of geometric shapes which could be used to create partitions.

- Polygons
- Rectangles

A rectangle is a special case of a polygon so all possible mentioned shapes can be dimensioned as polygons. Therefore an algorithm is presented which determines which polygons cover the requested area which is a polygon as well. The algorithm would be a bit more easy if only rectangles would have been used but this is not important for now. This solution is only presented to show that it is possible to find the right partitions in a simple way. There are many other options possible to find the right partitions and this could be a research project on its own.

The following function shows how a set of partitions which cover the requested polygon can be determined (the circle around the car or the corridor a driver will visit).

```
function Polygon[] PolygonsCoverPolygon(Polygon input) {
  // Array of polygons for the result
  Polygon[] result;

  // Take the first line of the input polygon
  LineSegment line = input.firstLine();

  // Find a starting point
  Polygon poly = FindCorrespondingPolygon ( line.point[0] );
  // find polygon which covers the first point
  // of the first line segment

  // Add the polygon to the result set
  result += poly;

  do {
    while ( ! PointLiesInPolygon(
                  poly, end point of line ) ) {
      result += poly;
      // Add the polygon to the result set
      // (assuming it is not added)

      result += PolygonsCovered (input, poly);
      // Add neighbouring covered polygons

      poly = neighbour polygon which is entered by the line
             when the current polygon is exited;
    }
  }
  while( line = input.nextLine() );
  // Repeat this for all lines of the input polygon

  return result;
}
```

The function starts with an empty result array which will be filled and returned. The first line of the input polygon is taken and it is decided in which partition the first point of that line segment is located (the function FindCorresponding-Polygon is responsible for that and it is described below). This partition will serve as a starting point from which on the neighboring partitions are investigated. The line segments of the input polygon are taken one by one and it is checked for each line which partitions are crossed. All those crossed partitions are added to the result set and together they are all the partitions which are partly covered by the input polygon. The partitions which are completely covered are added by the function PolygonsCovered which is described later on. At the end of this function a complete set of partitions is calculated and it can be returned.

In figure 17 on the following page a simple example is given of the algorithm just described. The small squares are the partitions and the input polygon is the gray rectangle. The starting point is the top corner of the rectangle which is located in partition 'a'. This partition is added to the result set. The algorithm

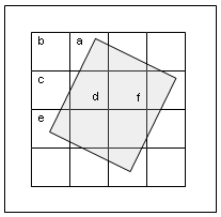

Figure 17: Partition coverage algorithm

checks if the end point of the line is located in partition 'a'. Because it is located in partition 'e', the algorithm will check all the neighboring partitions if they are covered by the polygon. This is not the case for any of the neighboring partitions of 'a'. Once this is done, the algorithm will find the neighboring partition which is entered by the line when it exits the current partition. In this case this is partition 'd' and here the algorithm will do the same thing. Now it will find a fully covered neighboring partition ('f') which is added to the result set as well. The algorithm will 'follow' the line until it reaches partition 'e'. Here it will determine that the end point of the line lies within this partition. The end point of the line equals to the begin point of a new line so the algorithm can do the same for each line until it reaches the begin point again. The result is a set of polygons which cover the input polygon.

Although all functions of the algorithm are explained by word, not all of them are as obvious. Therefore the functions are explained in the remaining of this section.

The algorithm above uses the function FindCorrespondingPolygon to determine in which polygon a certain point is located.

```
function Polygon FindCorrespondingPolygon (Point point) {
  for each ( polygon in layer ) {
    if ( PointLiesInPolygon ( polygon, point ) ) {
      return polygon;
    }
  }
}
```

This function takes the partitions one by one and uses the function PointLiesIn-Polygon (described later on) to check if the point is located inside the polygon. If the right polygon is found, then it is returned.

The algorithm is quite simple but it takes some computational time to determine which polygon is suited. This could be enhanced by adding some kind of bounding box around each partition. It is easier to determine if a point is located inside a rectangle then inside a polygon. For a rectangle one only has to take (e.g.) the upper left corner and the lower right corner. A simple check if the coordinate of the point is between both points is enough to conclude that the point falls within the bounding box. The algorithm then still has to check if the point is located inside the polygon but this, more complicated, algorithm is no longer executed for each polygon.

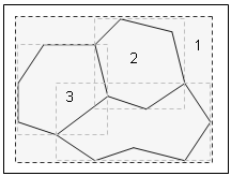

Figure 18: Bounding box example

A set of bounding boxes can be put together in a new bounding box which makes it possible to create a tree of boxes. The level of the tree determines the size of the area which is covered by the bounding box. To find the right partition, one has to start at the highest level of the tree and look at each level of the tree which leaves are of interest (the point should be within the bounding box of this leave). The tree under these leaves should be invested again which will finally result in the right partition.

This is explained by a graphical example which is shown in figure 18. Three polygons are drawn and every polygon has a dashed bounding box around it. These three bounding boxes are boxed again by a bigger bounding box which has a gray background color. The three numbered locations in the picture are all covered by the big bounding box but this does not always means that one of the inner partitions is actually covering the location (e.g. location '1'). The locations '2' and '3' are both covered by inner boxes where location '3' is even covered by multiple inner boxes. Each of these inner boxes have to be checked again to see which one actually covers the location.

The function PolygonsCovered is also used by the algorithm discussed before. This function is described below and it is used to search for neighboring partitions which are also covered by the input polygon. The previously described algorithm only finds polygons on the border of the input polygon and this algorithm is used to find the remaining ones which are fully covered. A small side affect is the fact that this algorithm can also find some partly covered polygons but this is not a problem.

```
function Polygon[] PolygonsCovered(Polygon cover,
                                   Polygon start) {
  // Array of polygons for the result
  Polygon[] result;

  // Check each neighbouring polygon
  for each ( neighbour polygon of start ) do {

    // Check if a random point in the polygon lies in
    // the covered input polygon
    if (PointLiesInPolygon ( cover,
                  point of neighbour polygon ) ) {

      // The point is covered, so the polygon is covered
      result += neighbour polygon;

      // Search each neighbour of this polygon as well
      result += PolygonsCovered ( cover, neighbour polygon );
    }
  }

  return result;
}
```

The function gets an input polygon (e.g. a corridor) and a starting polygon which is used as a starting point. This last polygon is a partly covered polygon and this algorithm checks if there are any neighbours which are covered as well by the cover polygon. This is done by taking a random point (e.g. a point of one of the line segments) for each neighbouring polygon. The function PointLiesInPolygon (explained later on) is used to check if the given point is covered by the input polygon. If this is the case, then the neighbour is added to the result set and all the neighbours of this neighbour are investigated as well. The result is a set of partitions which are covered (partly or fully) by the input polygon.

The function just described finds at least all partitions which are covered completely because every location in those partitions is covered by the input polygon. These partitions are typically not found by the previously explained part of the algorithm which uses the borders of the input polygon to find partly covered partitions. Some of the partly covered partitions are however found as well by this algorithm because the chosen point for these polygons could be located in the covered area.

This algorithm is graphically explained as well (see figure 19). Again the small squares are the partitions while the input polygon is the light gray rectangle. The function PolygonsCovered is called and partition '1' is the start point. From there all the neighboring partitions ('2') are investigated and two partitions are covered ('+' sign is used for that). For these partitions the function is called again (recursively) and again the neighboring partitions ('3') are evaluated. This goes at least on until all fully covered partitions are found. The result of this function is returned back to the algorithm described above.

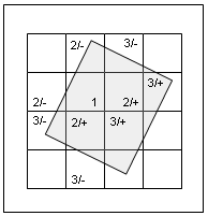

Figure 19: Polygon coverage explained

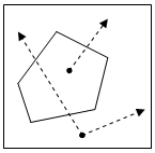

Figure 20: Ray casting

The last unexplained function is used several times within the algorithms and it can be used to check if a point is located inside a polygon.

```
function boolean PointLiesInPolygon(Polygon poly,
                                    Point point) {
  rayCount = number of times a ray, starting from the point,
             intersects with the line segments
             of the polygon;

  if ( rayCount is odd ) {
    return true;
  }
  else {
    return false;
  }
}
```

This function checks the partition by using a ray casting algorithm to see how many times the polygon is intersected with the ray. If the ray is intersected an odd number of times by the polygon, then the points lies within the polygon. Otherwise, when the ray is an even number of times intersected, the point lays outsides the polygon.

This is explained by a simple example in figure 20. A polygon and two points are visualized, one point within the polygon and one outside it. A ray from a point outside the polygon will always intersect the polygon an even number of times (0 or 2 in this example). The point located inside the polygon will always intersect the polygon an odd number of times.

The algorithm described in this section, results in a set of partitions which cover the input polygon completely. Some of the partitions will only be covered partly by this polygon while others are fully covered. As mentioned before, this

algorithm shows that it is possible to determine which polygons cover another polygon. Further research might show that it is possible to do it even faster.

5.2.6 Version control

In section 4.1 is mentioned that ActMAP already mentioned that it was possible to keep version control in more then one place:

- In the navigation system
- In the service centre

Both places have their own benefits in having version control in that place. This section discusses both options and finally one of them is chosen for our solution. We will first discuss the advantages of having version control within the navigation system. Based on these advantages we can already conclude that it is necessary to have it inside the vehicle. Then we will discuss the drawbacks and we will provide some solutions.

Having version control in the navigation system offers the following advantages.

- If the navigation system does not keep any version control inside the vehicle, then it does not know at all which version each partition has. This could introduce a problem for a navigation system if it wants to use this information to determine when to update a partition. There are some possibilities how and when to update partitions, these possibilities are discussed below.
  - Every time the vehicle is going to enter a new partition, that specific partition is updated. This looks like this.

    ```
    while ( true ) {
      if ( enters a new partition ) {
        update partition;
      }
    }
    ```

  - Once a destination is chosen and the route is calculated, the partitions which are crossed during the route are updated (see pseudo code below).

    ```
    if ( route calculated ) {
      for each ( partition which is crossed ) {
        update partition;
      }
    }
    ```

  - Based on one of the previous options, the system could also use the version information (assuming it is available). If a partition is just recently updated, it does not need to be updated again for some time

(might be configured by the driver). The example below shows this in combination with the second option.

```
if ( route calculated ) {
  for each ( partition which is crossed ) {
    if ( partition is not recently updated ) {
      update partition;
    }
  }
}
```

This last option is interesting because the information on POIs is not so volatile that a partition needs to be updated every time. The navigation system can use the version information so this information then must be available somehow in the navigation system.

- There is also a problem when a navigation system tries to install a minor release and the complete version control is in the service centre. The navigation system then needs the version information in order to update the in-vehicle database correctly. If there is already some newer information available in the vehicle, then the data should not be updated (already explained in section 5.2.2). So the navigation system should have all the version numbers in order to update the system from a minor update. Getting those version numbers from the service centre into the vehicle can take quite some effort and it is questionable if this is the behavior which is expected. One needs to contact the service centre (in order to get the version numbers) before one can install a minor update. This introduces a dependency between the offline update and the service centre which is not the intention.
- There must be support for multiple media to update the navigation system. It must be possible to update the system by both a wireless network connection and a memory stick. This last option does however never guarantee that an update (which is put on the memory stick) will really get into the car and is installed. A driver might for example forget the stick or loose it. When version control is within the service centre, a partition could be updated (within the service centre) without any guarantee that the in-vehicle database is actually updated with the update which is put on this stick. This allows situations where the in-vehicle database gets inconsistent because the version numbers at the service centre do not match the actual state in the in-vehicle database any more.

The last reason makes it necessary to keep version control within the car. Otherwise the consistency of the in-vehicle database cannot be guaranteed. Therefore it is chosen to keep version control completely within the vehicle and not in the service centre. There are however also some drawbacks for this approach. These drawbacks are discussed below and for some points a possible solution is presented.

- Version control inside the service centre would make live more easily for the navigation system because it would not have to take care for it. It will

reduce the required amount of disk space because the version numbers of each partition do not have to be stored within the vehicle but they are stored within the service centre.

Having version control in a navigation system means that there is some more disk space required to store the version information. The maximum amount of required disk space (s - in bytes) is equal to

$$Max\ s(n) = 4n$$

where n is the number of partitions over all layers. Because the version number is a timestamp (according to ActMAP), it needs at most 4 bytes for each partition. This can be reduced by using some more intelligent version management inside a navigation system. There are two points which can be taken into account.

- Most of the partitions are only updated by an offline update. It is expected that only a small part of the partitions are actually updated by an online update. This because a driver, which has installed for example a database for Europe, will (in most cases) not 'visit' all partitions. Most drivers are also just interested in a subset (some categories) of all POIs. In total there will be only a small part (estimation $<10\%$) of partitions which is actually of interest for the driver. The other unaffected partitions all have the same version number (of the last installed minor update) so one could only keep the version numbers of the partitions which are affected by the online updates.
- The POIs which are presented by a TPD supplier can change every now and then. It is however not the case that they will update the same data (in a partition) very often, the data is not that volatile. Therefore a timestamp might be too accurate. Instead of this one could agree on the fact that every update supplier can only update a partition (for example) once a day at the service centre. It is then possible to use version numbers where every day the version number is incremented by one. The navigation system could use day of the release of the baseline map as a starting point and calculate an offset from there. The 4 bytes which were required for each version number can be reduced to 2 bytes without loosing any time information.

Taking both points (and estimations) into account, it is possible to reduce the required size of the version numbers to

$$Max\ s(n) = \frac{n}{5}$$

which is 20 times smaller then before. This is because of the fact that not all version numbers are stored (10 times) and the fact that an offset can be used (2 times).

- By having the version numbers within the vehicle it is also necessary to send the version number of each requested partition to the service centre. The amount of traffic will increase a little by this compared to a solution

when version control is in the service centre (when no version information has to be transmitted). This small drawback is inevitable and there is no solution presented for this problem.

- Because the vehicle has all the version information in it, it will have to determine which partitions must be updated. If version control is within the service centre, then the navigation system can let the service centre decide which partitions must be updated. The navigation system then only has to decide which area must be updated (a polygon or circle) but this must be decided in this situation by the navigation system. More on this problem is discussed in the previous paragraph.

It can be concluded that version control has to take place inside the vehicle instead of having it inside the service centre. By having version control it is possible to update a partition correctly. It is however not mentioned yet how a POI should be addressed within a partition. This is discussed in the next section.

5.2.7 Identifying a POI

Until now it is only possible to identify a partition. Every partition can however cover more then one POI so updating a partition does not immediately mean that a POI is updated. Therefore a POI has to be identified within a partition. ActMAP has defined two ways to do this by using identifiers or by using location referencing. This section discusses both techniques.

The simplest way to identify a POI is by using a unique identifier for each POI. This identifier is added to each update on that specific POI. This means however that every POI needs an own unique number which has to be stored within the vehicle. This can take quite some space. If a database contains for example one million POIs and every POI needs its own identifier, then there are one million different numbers which each need three bytes to be stored. In total the identifiers then need around three megabytes. For a large database (1 GB) this is not that much but for the smaller databases (which can contain one million POIs as well but have less information on every POI), this can differ much.

Therefore it might be interesting to use location referencing in stead of an unique identifier. The location of a POI can be used to identify it. ActMAP already concluded that this technique is more error prone when for example AGORA [5] is used. ActMAP then expects that 90% of the updates are matched with the right POI inside a navigation system. By using the location (coordinate) as a reference, this number is not that low but it is still not guaranteed that the right POI is updated all the time. This is not in line with the requirements so some solutions are presented such that location referencing could be used as a solution which is also usable.

- Because the location of a POI is used as the identifier, all update suppliers should use a way to identify a location which is supported by the navigation systems without any conversions. A navigation system should store

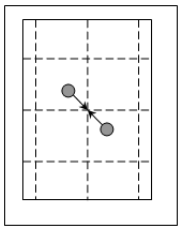

Figure 21: Location referencing

the location without any loss of precision. If one of both things happens, then it might be impossible to find the right POI if one is updated.

This is visualized in figure 21 where a large grid visualizes the points which can be stored within the navigations system. Two POIs are shown in this figure as grey dots on the location provided by the update supplier. If the in-vehicle database does not store the POIs with the precision which is provided by the update supplier, then both POIs are stored on the same place in the grid (see arrows). The same thing could happen if location values are converted and their location is somewhat changed.

- If the previous point is used, then a solid mapping for the locations provided by an update supplier to the locations inside a navigation system is guaranteed. It is however still possible to have more then one POI on the same location. A hotel and a restaurant could be placed in the same building without having any relationship. A TPD supplier could therefore have two (or more) POIs which have the same location. It is still not defined how these POIs can be identified (uniquely). There are two ways to do this.

  - Adding some extra information specifically for the identification which makes the POI unique in the location.
  - Using other information (data from the POI) to determine its uniqueness. For example the name of the POI or some other information.

  The first solution is preferable. An update supplier can determine how many POIs there are on a certain location and for each location which has more then one POI, it can give the POIs a simple number. This number is stored as an extra (hidden from drivers) attribute of the POI and it identifies each POI on that location. Besides the fact that most POIs do not share their location, this attribute is also very small in the vehicle where it uses not much disk space.

  The second solution requires less disk space but the update supplier has to determine how it can distinguish the POI (compared to the others). This is somewhat more complex and therefore it is chosen to use the first technique.

There still is a small problem when the location of a POI is changed. In that case the unique identifier changes. Update suppliers have to take care of this

and it is not expected to give any problems for them. A navigation system has to take this into account as well. They could use the unique identifiers outside the database to store for example the favourite POIs of the user or the unique identifier is used as a destination point. If the database is updated and the unique identifier is changed, then the navigation system should deal with it in a correct way. This is however a small point of attention when location referencing is used.

Using the solutions presented above, it is also possible to use location referencing instead of unique identifiers. It is then possible to identify the right POI all the time. The solution saves some disk space inside a navigation system but it is also somewhat more complex for both update suppliers and navigation systems. If size is not a real problem, then it is advisable to use permanent identifiers.

5.2.8 POI referencing

POIs are al the time described as stand alone points. For some POI products it is however possible to point from one POI to another one. A shopping mall can for example refer to parking lots in the vicinity where the driver can park the car. These references are stored as information inside a POI. In current databases these references do not cause any problem but when the in-vehicle database is updatable, then there could be a problem. A POI could have a reference to another POI which is not in the database.

This problem can be solved by adding dependencies between those POIs (and their partitions). If a POI refers to another POI then they can be updated at the same time. The change of having more dependencies increases however which increases also the chance of having the snowball effect. Therefore it is decided not to use dependencies in these situations. This will however result in situations where a POI points to another POI which is not up to date or which is (for example) not even present in the database. The software has to take care for these situations where it might be possible to point to a POI which does not exist.

5.2.9 Data dictionary

An update supplier has to know how it can update the data on POIs and a navigation system needs to know which data it can expect from the service centre and minor updates. They have to make some kind of agreement on the way the data is exchanged. ActMAP has introduced the data dictionary for this. Each layer can have an own data dictionary which describes how the data in that specific layer can look like. The update supplier must take care that all updates on a specific layer are compliant to the data dictionary. A navigation system then knows which data it can expect and how this data should look like.

The data dictionary is mentioned by ActMAP but it is not clearly explained in the documentation how this should work. It is important to have an agreement between update suppliers and navigation systems on the look of the data. Without those agreements, navigation systems will not know what they have to do with the provided data.

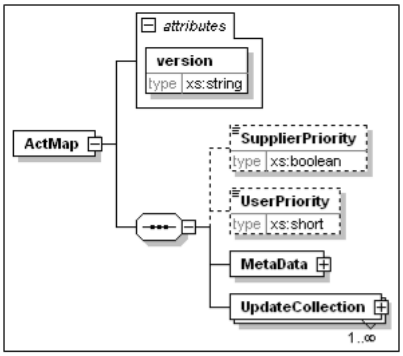

Figure 22: Update exchange format (1)

The data dictionary is however not been part of this research and further research is needed to work this topic out.

5.2.10 Conclusion

The presented conceptual model shows that it is possible to update the in-vehicle database by using the ActMAP concepts. These concepts are worked out in more detail and all options left open by ActMAP are filled in. Some options leave space for more possible solutions and these solutions are described in detail. The conceptual model is validated in chapter 6.

5.3 Exchange formats

ActMAP has presented several exchange formats which are based on the concepts presented by them. This section discusses the exchange formats briefly to show that the presented exchange format cannot be used without any adaptations. It is not meant to provide an alternative for the exchange format but it just evaluates the provided formats.

There are four exchange formats introduced by ActMAP. One of these formats is about the data dictionary. Because this was not part of this research project, this format is not discussed. The other remaining parts are shortly discussed in the remaining of this section to see if they can be used.

5.3.1 The update exchange format

Getting updates from one to another (e.g. the update supplier to the service centre) is done by the exchange format. We will show that the format provides a good start but we will also show that some adaptations are needed before it can be used. We will present a small part of it to show the most important drawbacks.

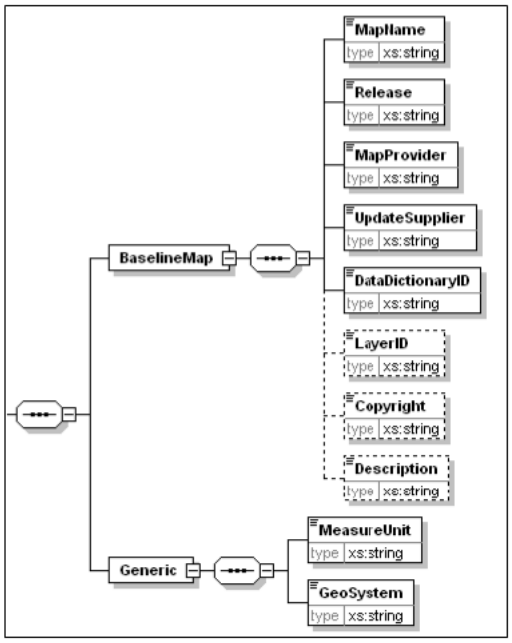

Figure 23: Update exchange format (2)

The root node 'ActMap' of the format is shown in figure 22 on the preceding page. There is a separation between some metadata and the actual updates which are stored in the UpdateCollection nodes. Every update collection (an ActMap node can have multiple of such nodes) has an attribute which is used to bind the update to a certain partition. Because of the rather late introduction of layers, it was decided to bind the layer identification to this same attribute. This binds the layer and partition identification together where it would be more logical to keep them separated.

The metadata node is shown in figure 23 and it contains some data which is applicable for all updates in the UpdateCollection nodes. There is a separation between the baseline map information and some generic information. The baseline map contains also information on the layer but this information is not separated from the baseline map information. It would make sense to separate the baseline map and layer information to make the format more transparent. By doing this, the format would better suit the conceptual model. Some information on a layer is stored inside the UpdateCollection node. This makes it also less transparent where all information is stored. Therefor it is advised to keep all this information together as much as possible in the MetaData node.

The measure unit and geo system were introduced in the generics part of the metadata for the first time while they were not discussed in the conceptual model. There was no reasoning found in the ActMAP documentation why they should be placed here. For example the time zone and the accuracy can be set for each update collection individually where this was not possible for these values (measure unit and geo system) which have to be set for an entire set

of updates. For these values it might be handy to set them for each update collection separately as well. For example, an update supplier can then supply updates for the UK and the Netherlands in the same update. The supplier can then use a different measure unit (inches) for updates in the UK while it can use meters as the measure unit for updates in the Netherlands.

The current update exchange format is not in line with the conceptual model presented in section 5.2. Therefore it is advisable to change the format so it becomes somewhat more consistent to the presented conceptual model.

5.3.2 The update request format

This is the format which is used by a navigation system to get updates or other information from the service centre. A navigation system can put several things in such a request, one can specify the baseline map, a filter and an ID for registered services.

The baseline map will normally be equal to the baseline map in the system itself so the service centre knows which baseline map the navigation system has. A navigation system can also check if the given baseline map is supported by the service centre. If the baseline map is not supported, a navigation system has no use to ask for updates.

The filter can be used by the navigation system to filter out the updates which are of interest. This can be done in several ways.

- The filter may contain one or more partitions which must be updated. The identifier of a partition contains not only the partition but it also identifies the layer. It would however make more sense if we just look at the hierarchal structure of the layers and partitions so one can specify the partitions in a layer when a request is send to the service centre.
- A navigation system can specify the duration for the updates. This option is however not of interest for our solution.
- An area can be specified so the service centre can update this area. Because version control is within the vehicle, the navigation system has to decide which partitions must be updated. Therefore this filter option is also not of interest for our solution.
- The filter also provides a way to get specific updates in the navigation system by asking for specific attributes. A navigation system can ask this way for very specific updates but this is somewhat inconsistent with the other ideas. The updates are nicely grouped and it does not make any sense to ask for a subset of this information. Therefore this option is also not of any interest.

The ID for registered services can be used by a navigation system to register certain services for a simple retrieval of certain updates. This option is however not requested so it is also not of any interest.

Many things in the update request format are not of any interest for this solution so they could be left out. The format is set up such that many combinations are possible but not all combinations have a meaning. Therefore it might be wise to develop a more robust request format which is more strict and which does not allow combinations which are not valid.

5.3.3 The update notification format

The update notification format is used by update suppliers to notify the service centre that new updates are available. The service centre then has to contact the update supplier in order to get the updates. This means that the service centre must take action to get the updates. This format could be used but it is not documented by ActMAP why this should be used. Another option, which is not discussed by ActMAP, is the option to let the update supplier send the updates to the service centre. Once the update supplier has new updates available, it could send them to the service centre so the service centre could store and register the new updates. This could be used as well but it should be looked into more closely.

6 Validation

The conceptual framework which is presented in the previous chapter provides a way to get updates in the vehicle. In this chapter, this framework is validated for both the navigation system side and the service centre side. For both sides the use cases from section 3.1 are used to validate the framework. The navigation system side is validated by conceptual algorithms and the results are explained in section 6.1. The service centre is validated by an implementation of which the results are discussed in section 6.2. The last section of this chapter deals with the conclusion of both validations.

6.1 Validation for a navigation system

The validation of the navigation system side is done by describing the algorithms which are needed. The pseudo code, which is already introduced in section 5.2, is used to describe the algorithms. For the navigation system side it is possible to validate the offline updates as well as the online updates. Both use cases are validated in the remaining of this section.

6.1.1 Use case 1

In this use case, the offline update is executed where a navigation system is updated by a disc. There are two situation described in the use case when this should take place. When the vehicle leaves the garage for the first time and when the driver visits the garage for a quick check. Both updates can be handled by the following algorithm which is explained below.

```
if ( disc with updates is inserted ) {
  if ( current baseline map is equal to baseline
       map on disc ) then {
    if ( current installed release is older then the one
         on disc ) then {
      if ( current installed release is supported by a
           delta on disc ) then {
        update_using_delta();
      }
      else {
        update_using_full_db();
      }
    }
  }
  else {
    copy_full_db();
  }
}
```

Once the disc with updates is inserted, the algorithm will check if both baseline maps are equal. If this is not the case (e.g., when there is no database installed yet like in the first situation in the use case), then it is assumed that the disc

contains a baseline map which should replace the already installed baseline map. If both baseline maps are equal, then the navigation system will check if the inserted disc contains a newer version. Having the same or an older version has no effect on the in-vehicle database and the update is finished. For a newer version (which is most likely) the navigation system has to look if there is a delta available which can be used. If there is one available, then the system should use it (so the driver can leave the garage quickly like in the second situation in the use case). Otherwise it should use the full database which is always present on every disc.

In total there are three different ways to update the in-vehicle database. The previous algorithm shows which of the three algorithms should be used in order to update the in-vehicle database correctly. Each of these algorithms is described below in pseudo code and a small explanation is added.

If the full database has to be installed into the navigation system, then the navigation system should first try to remove the current database. Once it is removed, it can install the new database which can be found on disc like this.

```
function copy_full_db() {
  try to remove current database;
  if ( the in-vehicle database is removed ) then {
    install new database into navigation system;
  }
}
```

If the baseline maps are equal and the disc provides a usable delta, then the navigation system will have to check all the data inside the navigation system against that delta and all old information must be updated. This is done by the following algorithm.

```
function update_using_delta() {
  for each ( layer in delta ) do {
    for each ( partition in layer ) do {
      if ( version of partition in current db is older
           then in minor release ) then {
        update partition to minor release;
      }
    }
  }
}
```

When there is no suitable delta, it is necessary to compare the full version on the disc against the in-vehicle database. The algorithm to do so is almost the same as for the delta but it is nevertheless shown below.

```
function update_using_full_db() {
  for each ( layer in full database ) do {
    for each ( partition in layer ) do {
      if ( version of partition in current db is older
           then in minor release ) then {
        update partition to minor release;
      }
    }
  }
}
```

These algorithms provide a way for the navigation system to update the in-vehicle database by disc. This was requested in the first use case and it was one of the two functional requirements. Both situations from the use case are handled by the given algorithm which proves the correct working.

6.1.2 Use case 2

The online updates from the framework have to take care for the situation in use case 2. Here a driver contacts the service centre to get updates on camping's in the vicinity of the car. The following algorithm can handle such an update once the driver asks the navigation system to look for updates. The algorithm is divided in several parts and each part is explained directly afterwards.

```
Polygon corridor = determine the area which needs
                   to be updated;
// this is a circle in this case which is converted
// to a convex polygon

// Partitions is an empty array which will be filled
// with all the partitions which need to be updated
partitions = array();

for each ( layer which is set to be updated ) {
  // Use the algorithm from section 5.2.5 to find the right
  // partitions which cover the corridor,
  // add them to partitions
  partitions += PolygonsCoverPolygon ( Polygon corridor );
}
```

In the first place, the system should determine a polygon which covers the area which must be updated. The driver is looking in the vicinity of the car so a circle can be put around the vehicle. This circle can be converted to a polygon which sets the area which needs to be updated. The driver might have selected the layers (categories) which are needed to be updated or these layers are already programmed before. For each of these layers, the system has to decide which partitions need to be updated. This is done by using the algorithm which is described in section 5.2.5. Once the system has calculated this, it has a set of partitions which need to be updated.

This set of partitions can be checked for any recently updated partitions. These partitions can be left out.

```
for each ( partition in partition set ) {
  if ( partition is recently updated ) {
    remove partition from partition set;
  }
}
```

This part of the algorithm is optional and it can be used to limit the number of updates on each partition. The in-vehicle system could check if the partition is recently updated. If this is the case, then it could decide not to update the partition which is then removed from the set.

Once it is known which partitions should be updated, the system can start to update these partitions.

```
requests[] = create a set of requests which can be send
             to the service centre;

for each ( request in requests ) {
  // Send the request to the service centre
  send request to service centre;

  // The response which contains the updates
  response = response from service centre;

  for each ( layer in response ) {
    for each ( partition of layer in response ) {
      apply updates by version number;
      update version of partition in in-vehicle
                  to newest update for partition;
    }
  }
}
```

A navigation system has to create a set of requests which have to be send to the service centre. This set depends on the used format and it is most likely that it is just one request. The set of requests is send to the service centre one by one. The service centre will answer a request and the response is evaluated within the navigation system. The response can contain updates and these have to be applied on the in-vehicle database. The most logical way to do this is to find updates for every partition which means that every partition needs to be updated only once. If all updates are applied, then the in-vehicle database is updated with the newest information.

After updating the in-vehicle database, the driver has an in-vehicle database which is updated partly. This means that the driver has all up to date information

The result is an in-vehicle database which is up to date. The driver, from the use case, has the most up to date information on campsites in the vicinity of the car which allows him to find the best place to stay for the night.

6.1.3 Conclusion

Both use cases from section 3.1 are verified on a conceptual level and this verification shows that it is possible to update the in-vehicle database by inserting a disc with a new version of the database or by asking specific updates from a service centre. Both use cases are verified and it can be assumed that the conceptual design works for the navigation system side.

6.2 Validation for a service centre

The validation process for the service centre side did not consist of some algorithmic pseudo code. It was actually implemented to see if the framework would work. This section describes the findings of this implementation.

The service centre only plays a role in the second use case where it can be contacted by a navigation system for updates. The navigation system will send a request to the service centre which will typically contain a message requesting for some updates. The service centre will respond by sending a message which contains the actual updates.

A conceptual architecture of the service centre was created before the actual design started. This architecture is shown in the figure 24 on the following page.

Both TPD suppliers and navigation systems can contact the service centre using an HTTP connection. They can send an HTTP request which contains a XML file. This file can be an update (for a TPD supplier) or a request from a navigation system. An HTTP server inside the service centre listens to these incoming requests and it will handle them. The HTTP server passes the requests down where they are analyzed and handled (by request handling). Request handling has to find out what is inside the message and it should be handled correctly. It uses the XML processing and validation block which can check the XML against a XML Schema. If a request is correct, it can be handled by the request handling block in an appropriate way and it can use specific managers for it.

Each manager has a specific task and all together they form the actual heart of the service centre. They provide the basic functionality which is offered by the service centre. An administrator can be used to register a new update while it will be stored by a storage manager. Every manager can store or retrieve data from the service centre. They can use files and a relational database which is accessible using the database block.

The conceptual model (described above) is split up into several layers to keep the service centre as modular as possible. This is visualized in figure 25 on the next page. The managers (at the bottom) are separated by several interfaces, one interface for each manager. These interfaces describe the functionality which is offered by a manager. The information which is provided by a manager is also defined by one or more interfaces for each manager. Managers can also use or require information from other managers by using these interfaces. This way the managers are completely separated from each other and from the handling layer on top of the managers. The managers are made available by a manager

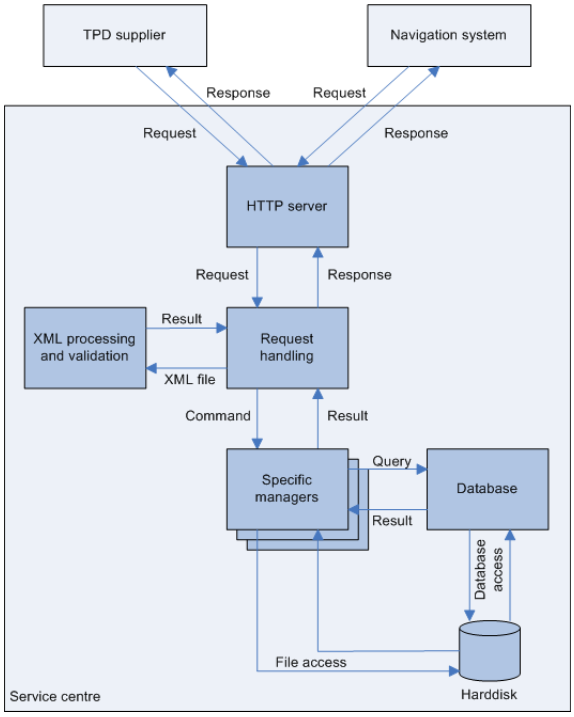

Figure 24: Architecture service centre

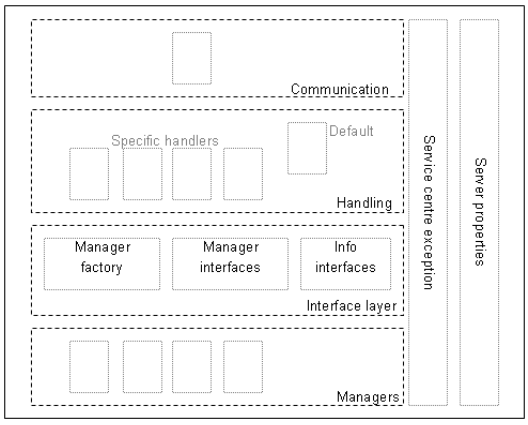

Figure 25: Layers in service centre

factory which can provide the requested manager to the handling layer or to another manager.

The handling layer contains several handlers which each can handle one or more specific requests. A default handler accepts all requests from the communication layer and it will find the right handler which will finally handle the request. The communication layer provides a connection to the outside world and it is responsible for handling HTTP requests. Every request is analyzed and correct requests are send to the default handler which can handle the request.

It is possible that something goes wrong within a certain layer. Therefore a general exception is created which can be thrown at all the layers except the communication layer. This is the only exception which should be thrown from one layer to another and all other exceptions are caught within the same layer.

There are some blocks which depend on something which is server specific. For example a directory which is used to store some data. A general block called server properties is made available for all layers and this can be used by any block if it needs some server specific information.

The result of this implementation is a service centre which implements the conceptual model which is described in section 5.2. The old exchange format is somewhat changed because it did not always fulfil the ideas of the conceptual design (as already stated in section 5.3). The results show however that it is possible to create a service centre in a simple way which meets the ideas of the conceptual design.

6.3 Conclusion

The conceptual design from section 5.2 is validated in this chapter by applying both use cases from section 3.1. The first use case (offline update) is validated on a conceptual level at the navigation system side while the second use case (online update) is validated at both the navigation system side and the service centre side. This use case is also validated on a conceptual level for the navigation system while the service centre is implemented.

In section 3.2.1 we already defined two functional requirements. The use cases are two typical examples which each represent such a functional requirement. They confirm the fact that both functional requirements, offline and online updates, are met by the design presented in section 5.

The non-functional requirements are stated in section 3.2.2 and we will look at each of these requirements to see if the design meets the requirement.

- An in-vehicle database should be consistent.

  Throughout the entire design, it is constantly pointed out that the in-vehicle database should keep its consistency. Options which could eventually cause an inconsistent in-vehicle database are not chosen because of their consistency problem. It can therefore be concluded that the in-vehicle database will be consistent all the time.

- Updates should keep the data as up to date as possible.

  Both offline and online updates are designed such that they keep more recent data in tact. For an offline update, each partition is checked before it is actually updated. An online update will always ask the service centre for more recent information which results in updates which only contain newer information. It can be concluded that this requirement is met.

- Use open standards as much as possible.

  ActMAP is an open standard on which the complete design is based. The design only uses concepts presented by ActMAP which makes the solution entirely based on an open standard. That makes that the solution also meets this requirement.

- Support for multiple kinds of update facilities.

  This non-functional requirement is also fulfilled by the presented solution. It is possible to use a wireless connection or one could use a memory stick or card as well.

- Limit the number of discs (CD or DVD).

  The number of discs is not necessarily increased by the solution although the amount of disc space required can grow due to the deltas which have to be put on a disc as well. This could result in some additional required discs but this will not always be the case (depends mainly on the size of the database).

- Updates by discs should be as fast as possible.

  Based on the amount of disc space left, it is possible to add deltas to a disc which can increase an offline update significantly. The amount of data which has to be copied from the disc to the system can be much less compared to a full copy. This makes an offline update much faster.

- Available memory (inside car) has to be taken into account.

  No research has been done at the navigation system side to determine the required amount of memory. Therefore it cannot be concluded that this requirement is fulfilled.

- Capable to determine the in-vehicle database status.

  The status of the database depends on the status of all individual partitions. A set of version numbers of all partitions in the database could provide an overall current database status.

- Extendibility.

  This was not one of the items which had the main focus during the design phase so it is not explicitely mentioned in the design. There are however always possibilities to extend the design as long as it is kept backwards compatible.

- Available bandwidth / costs to transfer data.

It is hard to verify if the required bandwidth is kept to a minimum. For communication, ActMAP suggests XML which is known to be large in file-size. Compression techniques could however reduce the size of an message which could solve this problem.

It can be concluded that the most of the non-functional requirements are met as well by the presented solution. Specifically the most important ones are covered which makes the solution fit the requirements from section 3.2.

7 Conclusion

This thesis tried to find an answer on the research question how to update POIs dynamically. We already concluded in section 1.1 that the answer to this research question could be found by answering the four questions mentioned in that section. We will use these four questions again to answer the main research question and to summarize the findings within this thesis.

1. What are the requirements for dynamic updates on POIs?

   The functional requirements for POIs were mentioned in section 3.2.1 on page 8 where two different ways were described on how to update an in-vehicle database:

   - Using an offline update - updating the in-vehicle database via a CD or DVD.
   - Using an online update - using a dynamic way (e.g. a wireless connection) to update the in-vehicle database partially.

   The provided solution should facilitate both ways and some non-functional requirements (section 3.2.2 on page 8) should be taken into account. The most important ones are:

   - The in-vehicle database should be consistent all the time.
   - Updates should keep the database as up to date as possible.
   - The use of open standards is encouraged in order to get the solution accepted as quickly as possible.
   - Online update should be possible by using different update facilities like a wireless connection or a memory card.
   - The number of discs for an offline update should not grow by the presented solution.

2. Is the ActMAP framework suitable for dynamic updates of POIs? If not, what is a good alternative?

   In section 5.1 it was discussed wether the ActMAP framework (summarized in section 4) could be used or not. It was concluded that the framework could be used because it provides ways to update the in-vehicle database in both ways as described in the requirements (offline and online). There are however some open issues and these issues are adressed and answered by the next question.

3. Given a certain framework, such as ActMAP or an alternative, which (design) choices have to be made to create a solution?

   In section 5.2 we described the conceptual design which provides a solution to update the in-vehicle database using both offline and online updates. A small summary is given to describe the most important design choices.

- For offline updates it was choosen to have a single disc which contains at least the complete database. This makes it possible to update the in-vehicle database always, regardless of its situation. The disc should also contain some deltas which makes it possible to update an in-vehicle database which is based on a prior minor (or major) update of the database with the same baseline map. A delta makes it possible to update an in-vehicle database quickly so a driver who visits the garage often enough (e.g. ones a year) will always have an up to date database as quickly as possible.
- Splitting POIs over different layers makes it possible to update the database partly so a driver can get updates on a subset of all POIs. Firstly it is possible to split different products over different layers (section 5.2.3). A driver then gets the possibility to update a single product at once instead of all at the same time. It is even possible to split the POIs inside a product over different layers as discussed in section 5.2.4 where we distinguished three ways.
  - Create layers which each store a complete (stand alone) set of POIs. If the same POI is represented in multiple layers, then the POI is stored in each layer separately. This is the major drawback of this approach because the same POI then can have different states within the in-vehicle database.
  - Use the same approach as described in the previous solution but now use cross-category layers to store POIs which are present in multiple layers. This solves the problem of having multiple states for a POI. The in-vehicle database needs however to update multiple layers and each updated cross-category layer could have POIs in it which are not of interest.
  - The POI information could also be stored in multiple layers which form a hierarchical set of layers. It is possible to use directed hierarchical dependencies to get all the related information into the vehicle when one of the top layers is updated. Putting information in different layers needs to be done carefully because dependent information within a POI needs to be stored in the same layer. This solution can however provide a nice and clean solution if everything is correctly chosen.
- Secondly it is also necessary to divide the POIs over partitions (section 5.2.5) so a driver does not get updates on POIs which are not in the vicinity of the requested area. It is recommended to use polygon shaped areas for the partitions. Calculating which polygons are of interest must be done by the navigation system so this might not be time consuming. Therefore one could use rectangular bounding boxes which reduce the number of partitions needed to be calculated.
- Every partition gets its own version number. This keeps the amount of disc space limited compared to a version number for each POI. The version numbers need to be stored inside the navigation system. This allows such a system to determine on its own which part of the in-vehicle database need to be updated. Offline updates can then be updated quickly without any interference of the service centre.

- One could use location referencing for the recognition of POIs in a partition (section 5.2.7) but we recommend the use of permanent IDs if disc space is not a real issue (e.g. large databases). The use of permanent IDs require some more disc space but it is much easier to implement and one can use the permanent identifiers for other purposes inside the navigation system as well.

All these design choises are in line with the conceptual ideas of ActMAP but we do not recommend the use of the exchange formats (see section 5.3). The initial formats have several inconsistencies like the partly integrated layering concept. Therefore we advise to revise these formats to find a solution which fits the concepts in a better way.

4. Is the proposed solution implementable and to which extend does it satisfy the requirements?

   The proposed solution is verified for both the navigation system side (section 6.1) and the service centre side (section 6.2). The verification on the navigation system side is based on the described algorithms and these show that it is possible to update the in-vehicle database on a conceptual level. The service centre is actually implemented and it shows that it is relative simple to create a solution. This answers the first part of the question, yes the solution is implementable.

   To finalize the validation, all requirements (functional and non-functional) are checked in section 6.3 to see if they are met. All the functional requirements and almost every non-functional requirement is met which brings us to the conclusion that the presented solution satisfies the requirements.

It can be concluded that it is possible to update Points of Interest in a dynamic way. It stays possible to update the database in the old way using discs but the presented solution also provides the oppertunity to update the in-vehicle database in a more dynamic way using online, dynamic updates.

References

[1] K. Ahmed. Mobile internet technologies and infrastructure. *IEP-SAC Journal*, 2003-2004.

[2] M. Aleksic J. Meier J. Löwenau M. Flament A. Guarise A. Bracht L. Capra K. Bruns H. Sabel H. U. Otto, L. Beuk. Actmap specification, 2005.

[3] A. Guarise M. Aleksic H. Sabel H. U. Otto J. Meier M. Flament J. Löwenau, L. Beuk. Final map actualisation requirements, 2004.

[4] M. Aleksic A. Guarise J. Löwenau L. Beuk J. Meier H. Sabel M. Flament, H.-U. Otto. Actmap final report, 2005.

[5] H. W. Pfeiffer K. Wevers T. Hendriks, H. U. Otto. Specification of the agora location referencing method, 2003.