1 September 1994

Modular handling of TMOS systems

Modulär hantering av TMOS system

Henrik Hansson
Olav Queseth

Abstract

This document will discuss how handling of TMOS may be simplified. The method discussed is to divide TMOS into modules. Since the size of the modules and the relations between them are important, a method for dividing TMOS into independent modules is proposed and discussed. A method of describing dependencies is also presented. This method allows a system to be resized and reconfigured when new modules are added. The method is intended to describe dependencies to mutual resources in the platform, but it can also handle dependencies to resources in other modules. All dependencies and engineering data are to be described in a ‘module description document’ that will replace the existing engineering data document. With this new document it is possible to detect problems when two or more modules need to share a resource. It also allows the new module to be tested in an optimal way. Since modular handling can be complicated, additional methods for simplification are introduced, e.g. installation procedures, a new AL concept and a preconfigured platform.

Contents

1 Preface

This work has been performed as a thesis project, which is a part of the Master of Science program in Computer Science and Engineering at Chalmers University of Technology in Göteborg.

There are two aspects of this work; the academic view of modular handling of large programs and the specific application TMOS at the SD department at EHS. This document is intended for readers both inside and outside Ericsson.

We have gathered the information through reading of appropriate literature and through interviews with people at EHS. The fact that the thesis project has been a part of a real project at EHS makes it challenging and more interesting. We are happy to have had this opportunity to learn about Modular Handling in general and Ericsson in particular.

1.1 Acknowledgements

We would like to thank all the people at EHS in Mölndal who have willingly answered all our questions and given us help all the way, especially our supervisor Niclas Nilsson who has had infinite patience (in between his own meetings) and has given us invaluable support in our work.

We would also like to thank our examiner at Chalmers University of Technology, Björn von Sydow, for his support and good advice.

2 Introduction

Today there are some problems in the handling of the TMOS product. The time used for handling has increased with each new revision of TMOS. If it shall be possible to handle TMOS in the future something has to be done.

This report describes a model to simplify handling. This is done by dividing the system into large blocks called modules and by describing dependencies using resources. By being careful when defining resources it is possible to achieve a good compromise between the amount of documentation and the amount of problems that can be detected.

This model is able to handle dependencies that result from two modules being dependent on the same resource. However, the model also has enough generality to handle dependencies where one module is dependent on a resource in another module.

To simplify handling even more standard configured systems can be created similar to the AL concept today. The standard systems are more standardized while they allow future expansion. The platform module is preconfigured so that it can be used by most combinations of modules without having to be reconfigured.

All solutions are discussed and motivated. The result is a module description document that is intended to replace the existing engineering data document. This document describes the resources in a module and which resources the module needs in other modules. This document is described in chapters 16 and 17.

This report is written from the viewpoint of the SD department. How modules will affect the work in design and other departments is only vaguely considered. In chapter 19 there is a description of which documents that need to be altered. This will indicate what has to be done in other departments to adapt to modular handling.

3 TMOS in a nutshell

TMOS is a software tool for handling all management and operations support within public telecoms networks. It acts as an ‘enabling layer’, sitting between the network elements and the network operation and management personnel. The network elements include switches, transmission systems, radio base stations, computers and other network management systems, of practically any type. Since TMOS can interface with every element in the network, the operational and management personnel can perform almost any task within the network.

The central core of TMOS is an object oriented database, which contains information on all the network elements. It provides a model that accurately reflects what is happening in the network. The model is continuously updated, gathering information form the network elements themselves, and implementing commands from the operators.

Today TMOS consists of five large subapplications, each with its own special task.

- XMs basic functions is sending commands, exchange files and supervising alarms to different parts in the network. It is also used to do performance measurements

- SMAS is used to implement and administrate different services within the network. The services could be e.g. letting the telecom switch act as a company switch, or administrating 020- and 071- numbers.

- CMAS is the subapplication that handles cellular mobile telephony.

- FMAS administrates and handles the telecommunication ‘highways’ in the network. It is used to find bottlenecks in the network and handling the correction of overloaded switches and ‘highways’.

- BMAS is used by large telecom customers to handle their internal usage and costs for telecommunications.

4 The structure of TMOS documents and programs

This section will describe the essentials of how TMOS is structured today.

At the top of the TMOS structural tree there is Application Lines (AL). These are complete systems that can be sold to a customer. They consist of several sales objects which is called FABs. Some of the FABs are optional in an AL and the other are mandatory. A FAB is a group of programs that is a stand alone application within the TMOS system. The relations between the FABs are a bit diffuse. There are design rules that says that each FAB has to be independent of other FABs, but these rules are hard to follow since dependencies could appear when two FABs uses the same functions in the platform (TMOS operating system). A FAB consists of one or many CXCs. A CXC is the smallest part that is handled in a TMOS system. Every CXC consist of a number of executable files, scripts and datafiles. It is at CXC level that the most documentation is made. Every CXC has its own documents for installation, engineering and specifications. At FAB level there not that much documentation. The documents at FAB level is mostly marketing and handling documents rather than function descriptions. At AL level there is more documents such as installation documents engineering data documents and user and reference manuals. These documents are derived from the documents at CXC level.

An AL is a general system, so when a customer orders a specific system an application system (AS) derived from the AL. If another customer wants a system

with the same configuration, this AS can be reused. In order to know which AS a specific customer has, a IPB document is generated.

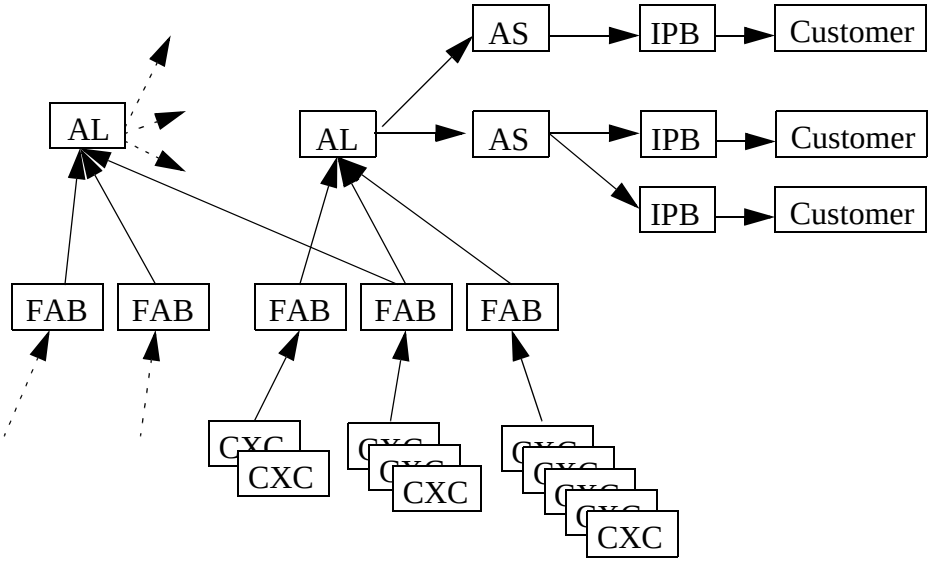

Figure 1. Structure of TMOS.

5 Current system

The handling of the system is today based on the application line (AL) concept. An application line is a standard product that consists of both mandatory and optional FABs and all the documentation that goes with them. An AL could for example be FMAS, SMAS or any other MAS for either a Sun or HP platform. An application system (AS) is then derived from the AL by selecting the mandatory and optional parts according to the requirements of the customer. The AS consists of software, installation and configuration descriptions, maps, and manuals specific to the customer.

Over the last few years TMOS has grown considerably. This growth has created problems in the handling and installation of the system. Due to the size of the system customers want to be able to buy parts of TMOS applications and exclude other parts that are not necessary for their needs. If the customer buys some mandatory and some optional parts from the same AL, the application can be sold without further testing. If a customer wants to combine functionality from different ALs there is no assurance that this customized application will work. They have to be tested separately, while undocumented relations between FABs renders default parameters invalid.

An AL is created by gathering information on the CXC level and compressing the installation and configuration information into an AL specific engineering data document, a parameter list and a maiden installation document. The time used to generate an application line is growing for each new release of the application. Time-consumption has now reached the limit of what is acceptable. The reason for the time spent in the generation is that information has to be gathered from many different sources since there are almost no documents at the FAB level. Another reason is that parts of the documentation have not been completed by Design at the time of AL generation, so assumptions are made. The creation of an AL is a sequential task, with one part of the AL being created after another. Because the AL is generated by people who are less familiar with the new functions in the application a great deal of time has to be spent on learning details. A lot of work is done by using CXC documentation. These documents are often very detailed and much of the information is irrelevant to the creation of an AL.

6 Problems with using AL

There are problems with handling the TMOS system the way it is done today:

- The AL has to be created centrally and it is hard to distribute work among people. Only when all components are ready can the AL be generated.

-The amount of knowledge needed to generate an AL is considerable. Only the most experienced people are able to do it since they have to have full control of the whole AL. In order to make an AL, one has understand all documents at the CXC level and know which people to ask questions.

- When adding functionality to an existing system, it is necessary to test large parts of the system even if no dependencies exist. This is the case when creating a special AS, due to lack of detailed knowledge of how the system really works. This kind of information is not easily extracted from the mountains of documents provided with TMOS and it is almost impossible to learn every dependency in TMOS by experience.

- If an AS differs from an existing AL, new engineering data and parameter list documents have to be written, for example if an AS consists of parts from two ALs or if a customer wants to exclude mandatory parts. Writing new documents is not a very difficult task, but it takes time and is for the most part unproductive.

-There is no proper documentation on which dependencies exist between different FABs. This makes it difficult to leave out unnecessary FABs when creating an AS. The customer therefore has to pay for functionality that is not wanted and perhaps buy more hardware than needed.

-The installation of an application is also very time-consuming. This is due to a lot of workarounds at installation time, answering a lot of similar questions and editing huge numbers of files just to change parameters, that could just as well be changed automatically. There are several reasons for this, but the main reason is probably lack of time at the release point and poor specification of how an installation script should be written. Installation and handling of CXCs have always had low priority, because one can make workarounds so that the installation “works” anyway and it is nothing that the customer notices.

- Errors are corrected at SD by introducing workarounds. It may happen that error reports are not sent back to the Design departments, and the errors will then remain in the next revision.

7 Advantages of using AL

- It is easy to install and configure systems that can be derived directly from the AL. A system derived from an AL does not have to be tested, since the AL is tested as a unit.

- Since the number of ALs is small, only a relatively small amount of documentation is needed.

- The installation is fairly simple. Only one document has to be used in order to make the installation.

- SD can make workarounds. That way, error corrections can be made quickly without the need to go through the Design departments.

- It is easy to have one person who is responsible for the whole AL. That person then has control of a complete system.

- An AL can be customized so that it suits the requirements of the customer. Each FAB can be given a unique configuration so that no extra capacity is used, and the customer does not then have to purchase more hardware than he needs.

8 Introduction to solution

In order to solve the handling problems, fundamental changes have to be made in the way that the TMOS system is handled today. There will of course be consequences that need to be considered.

The concept of creating an AL has to be changed. The SD department should not have to create documentation for the products that they install. Their time should be spent installing systems, not writing documentation for products already made. That means that the responsibility for documenting the application has to be moved to the Design departments. This will be more efficient because the people who write the programs have a more detailed knowledge of the inner working of the programs and should thus be able to create the documentation faster and more accurately than is the case today. This means that the people who perform the installation do not need to have so much detailed knowledge. But this will also require the installation procedure to be designed so that a TMOS system can be installed without detailed knowledge.

Today the various parts of TMOS are entangled in a way that is hard to overview. This is illustrated in figure 2. In order to ease handling problems the system has to be split into a few clearly defined parts. This is illustrated in figure 3.

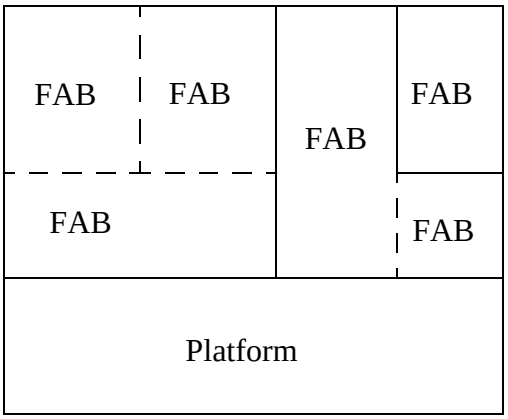

Figure 2. The TMOS system today

Keeping the possibility of configuring the system according to the requirements of the customer is important.

8.1 Dividing the system into modules

In order to decentralize the task of documentation a system has to be split into smaller parts, modules. By doing this, information on how the system fits together and which parts that match each other is lost. In order to be able to put the modules back into a system again the interface between the modules has to be described thoroughly.

It is important to be able to configure the system according to the requirements of the customer. Fortunately, the modules provide a good way of keeping this feature. If the modules can be installed separately, this can provide a good mechanism for customizing the system. Another benefit from being able to install modules separately is that the installation of each module can be done by different individuals. The amount of knowledge that each individual has to have can therefore be reduced. If new functionality is to be added that is put in a separate module. That way there is no need to redesign modules.

The platform concept helps in the creation of modules because it provides services that many modules need. The platform should be the ‘operating system’ for TMOS that provides all the services that the modules need. If the design of the platform is good the modules can be made more independent. The platform itself can be viewed as a special module. The platform should preferably not contain any optional parts that other modules use. An illustration of how a system may look tomorrow can be found in figure 3.

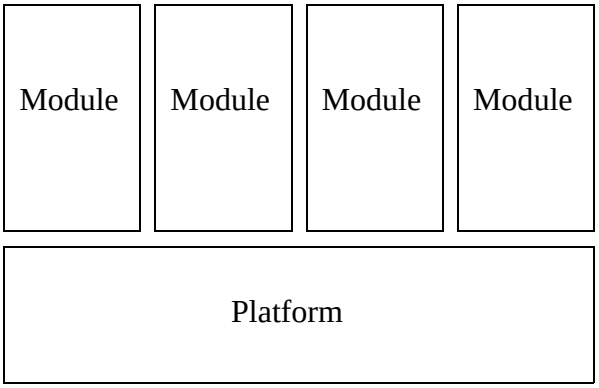

Figure 3. The TMOS system tomorrow

8.2 All dependencies are described using resources

All modules are dependent on the platform module and a few modules depend on other modules. This means that there has to be a mechanism for describing such

dependencies and handling any problems that may arise. To do this it is necessary to describe everything that a module needs to function correctly that is not present in the module, e.g. databases, CPU, disk storage and server processes. All such things are called resources. If this is done for every module and every little resource that a module needs is described, it would make detection of all problems possible, at least in theory.

Of course it is easy to realize that if all resources were to be described there would be enormous amounts of documentation. Fortunately, there are a couple of things that can be done to reduce the amount of documentation that has to be created and still be able to detect most problems that may occur. This can be done by not describing the resources that obviously have to be present in the system. Another way to reduce documentation is to ease the demand for an exhaustive description of all resources. Only a few resources can cause problems by being underdimensioned and it is only these that need to be completely described. A simplified description can be made for all the other resources.

All information that is needed about the resources is collected and put together with a description of a module and the information that is normally included in the engineering data documents today. This document then replaces the engineering data document. This document, a parameter list and an installation document will be the documents that are used when configuring and installing modules. The document should contain information on dependencies and resources and descriptions of how to reconfigure and rescale the resources when other modules require it.

8.3 An AL-like concept is introduced to simplify handling

By introducing a new concept on top of the modular thinking the workload will decrease even more. Since a group of modules are normally handled together it is possible to do all the calculations for them once and for all. The result of all the calculations is a parameter list. This parameter list can then be used again when the same modules are to be installed together. This is similar to the ALs used today but with two differences. There is very little documentation at the AL level, just a parameter list, and it is still as easy to add a module to a system as if all the modules were handled separately.

9 Modules

With the use of modules, handling of the TMOS system can be greatly simplified. The size of the module is, however, crucial to maximum simplification.

There are many factors that influence and are influenced by the size of a module. Since there are many issues that need to be addressed and since some issues have conflicting demands a trade-off has to be made. To acquire a more thorough understanding of the trade-off that has been made in this solution, the factors that influence size will be explained more closely.

9.1 Atomic objects

In order to simplify this discussion the concept of the atomic object has to be introduced. Note that an atomic object does not necessarily have to be a module. An atomic object is an object that can not be divided into smaller parts. A number of parameters and other configuration information are associated with each object. There is a way to find information on how an atomic object interacts with its environment. This information describes which resources are used, which other objects this object needs in order to function/install correctly and so on.

Consider a TMOS system that consists of a number of atomic objects. Of course the size of the atomic objects is related to their number in a TMOS system. The larger the objects the fewer there are. Note that an atomic object is not identical to a module. Modules will be introduced later.

When deciding the optimum size of an atomic object there are many factors that need to be taken into consideration. These will be examined a little more closely.

9.1.1 Configurability

The size of the atomic object influences the amount of customer configuration that can be done. If a TMOS application consists of many atomic objects many objects can be made optional, and thus allowing the customer great freedom of choice. Large atomic objects result in few objects in a complete system, which means less freedom of choice. Of course, objects that are necessary for the correct operation of the system can not be taken away. Another important fact to notice is that customers tend to choose functions that are related to each other.

9.1.2 Number of dependencies and amount of documentation

It is the dependencies between atomic objects that need to be documented. Since the total number of dependencies in a system increases with the number of objects the amount of documentation that is needed grows with the number of atomic objects. It is thus important to reduce the number of objects to minimize the amount of documentation.

9.1.3 Cost of handling

The amount of work and the cost that is associated with an atomic object are roughly the same for all sizes of objects. Included in handling costs are those costs associated with creating and maintaining tapes, writing documentation and so on. It must be possible to handle objects independently, otherwise they would not have been atomic objects, i.e. if two objects are always handled together, installed together and so on, they are really one atomic object. Separate handling implies that each object has to be configured independently. It is thus desirable to have as few objects as possible in a TMOS system.

9.1.4 Development

The parallelism in the development of a whole system increases as the number of atomic objects in a system grows. Development can be carried out on all the objects at the same time. There is a limit to the amount of parallelism that can be achieved though, since atomic objects could be dependent upon each other.

9.1.5 Revisions

The handling of different revisions of atomic objects is a complex issue. If objects are large, there is a lot of work associated with the creation of a new revision. A problem with small objects is that each atomic object may have several dependencies to other objects which may call for updating of several objects at the same time. If objects are stable and do not change much, it does not matter if an object is large since new versions are seldom created.

The same arguments that apply to handling in general can also be applied to handling of different revisions and thus the cost of handling revisions increases as the number of atomic objects in a system grow.

9.1.6 Comprehensibility

The understanding of the system increases as the number of objects in a TMOS system decreases. It is easier to get a general overview of the system if there are a few objects to keep track of, at least if the objects are logically connected, although with large objects the details of how things work is lost. The thing to keep in mind is that with large objects it is easier to get an overview of the system, but details must be sought elsewhere.

The table in figure 4 summarizes the different factors that must be taken into consideration when deciding the size of the individual atomic objects.

Parameter	Small objects	Large objects
Configurability	High	Low
Total number of dependencies in a system	High	Low
Amount of documentation	Much	Less
Cost of handling	High	Low
Available parallelism in development	Much	Less
Amount of work when creating a new revision / correcting errors	Low	High
Comprehensibility of system	Low	High

Figure 4. The size of the atomic object affects many parameters

9.2 Different ways of creating modules

Now the module concept is introduced. A module is a part of a system. There is an interface description associated with each module, describing which dependencies a module has to its environment.

There are many ways of dividing a TMOS system into modules. Four different approaches will be presented here:

9.2.1 Small modules without options

One approach is to let a module be about as large as an FAB is today, and let all the modules be atomic objects. Handling of the modules in this solution, however, involves a great amount of work and a very high cost. The problem is not the complexity in handling each individual module, but the large amount of modules that need to be handled. If handling of the modules can be simplified and the

associated cost can be greatly reduced, choosing small atomic objects can be a way of solving the problem.

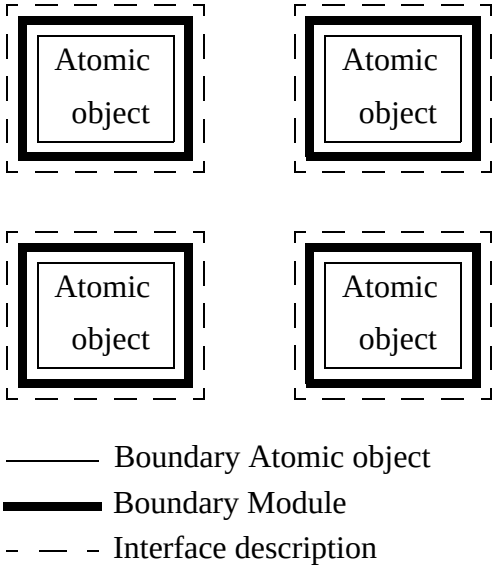

Figure 5. A system with small modules that are the same as atomic objects

9.2.2 Large modules without options

Another solution is to let the modules and the atomic objects be large, possibly the size of five to ten FABs. The problem with this approach is that the configurability of such a solution is low. In fact it would be lower than for an AL today. There are also problems associated with development and testing. Although a solution that yields low handling costs, it is not a feasible solution to the problem since it does not provide enough configurability.

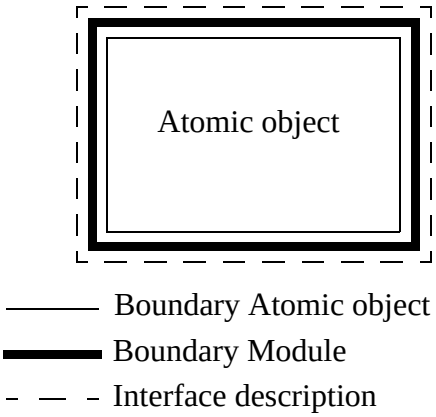

Figure 6. A system with large modules that are the same as atomic objects

9.2.3 Large modules with options

Yet another solution is to keep the modules fairly large but let the atomic objects be small. The atomic object here is approximately the same as an FAB. The module then consists of many FABs of which a few are optional. This is a solution that keeps the number of modules that have to be handled small and thus keeps the cost of handling down, while still allowing customers to make some options in their system. A problem that is introduced with this solution is that the modules can no longer be regarded as black boxes. This increases the complexity of the description of the individual module, illustrated in figure 7. Instead of describing dependencies upon other modules, dependencies between FABs have to be described. Resource requirements can not be described in a simple way for the module. Instead resource requirements have to be described depending on which FABs are present and so on. The nice thing about this solution is that it keeps the number of modules in a system small. The trouble is that the complexity in describing each module is increased a lot. An example of a large module with options would be an AL with FABs as atomic objects. This is not a great example though, since modules should be smaller than ALs.

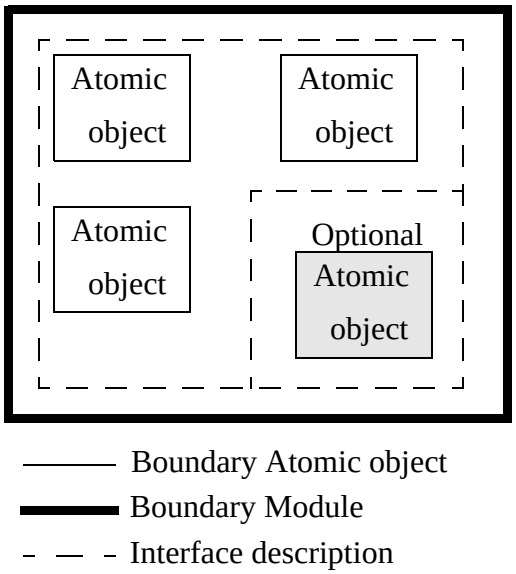

Figure 7. The interface description changes depending on if options are installed or not

9.2.4 Large modules with constant interface description

The fourth solution is to keep the module large, but the atomic objects small and let a few of the atomic objects be optional. An atomic object is approximately the same as a FAB. The only resource requirements described are those for a complete module with all options installed. This is illustrated in figure 8. With this solution the description of the module can be kept relatively simple. There are only a few things that need to be described at the FAB level. Most things can still be described at the module level. The drawback is that more resources, e.g. disk,

CPU power etc. than are really needed are reserved for the module. But this might not be a serious drawback since customers may want to add options to the modules that they already have and then all resources needed will be present. It is suggested that third party products are not made optional, since Ericsson has to pay for the license for this software even if the customer has not chosen the option. If an option uses hardware, it is of course necessary to be able to install the module without this hardware present at the time of installation.

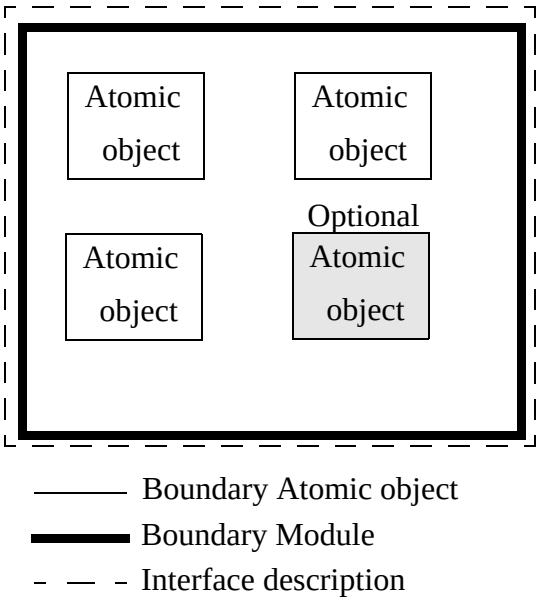

Figure 8. A module with options with constant interface description

9.3 Comparison of the various approaches

The first solution has the advantage that it offers a simple approach and that the handling of each module is much simpler than in the other solutions. The drawback is the number of modules that a system will consist of. The cost of handling and engineering will be too high. Engineering for a system will be as complicated as creating an AL today.

The second solution features low costs of handling and engineering. The drawback is that it offers few possibilities to customize a system. There would be only one or two options in a complete system, i.e. less freedom of choice than in an XM system today.

The third solution features few modules, which indicates that low costs of handling can be achieved. The drawback is that there are many details within the module that need to be described for example there will be a number of different dependencies that need to be described for all the FABs present in a module, how

much storage each individual FAB uses and so on. There is not much difference between the first and the third solution when considering the amount of handling and engineering required.

The fourth solution promises simplified handling and thus low costs. The drawback is that more resources than are actually needed are allocated for the module. However, it does seem the most viable solution, since it offers a way of handling the system without excessive cost. It is therefore given a more thorough description.

9.4 Creating modules

Creating modules is a complicated task since there are so many different factors that must be taken into consideration. All these factors need to be weighed against each other in order to create modules that can be handled at minimum cost.

9.4.1 Size of modules

There are many factors that influence the size of a module. Some factors result in low handling cost if there are a few modules in a system. Others give low handling cost if there are few FABs in each module.

Today a typical system contains some twenty FABs. If many FABs are included in each module there will be fewer modules in a system of any given size than if there are few FABs included in each module.

Figure 9 illustrates how the different factors influence handling cost. It is important to note that the figure is based on educated guesswork. There is no guarantee

that the figure is correct, but it should at least give an idea of how a correct size of a module might be chosen.

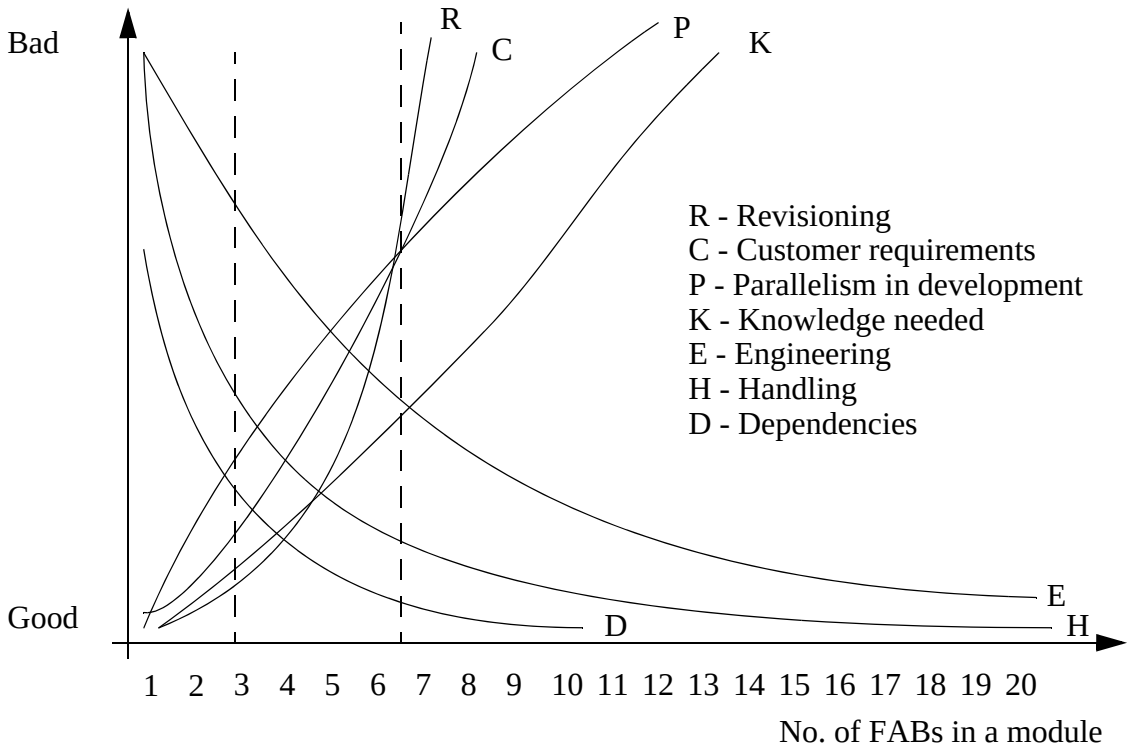

Figure 9. Influence of various factors on the size of modules

The figure indicates that three to seven FABs might be a good choice to make up a module. This would result in four to six modules in a system. In the future there might be even more modules in a system.

The best size for each module must, however, be decided on a per module basis by weighing the various factors against each other.

9.4.1.1 R - Revisioning

If more FABs are put in a module, revisioning and error correction will become slower and more complicated. Up to three FABs can be handled conveniently but after that the work required for a revision and the time it takes increase rapidly. It is important to note that a module that is stable and does not require frequent revisioning can be larger than a module that is revised often. This is because even if a revision requires a lot of work it does not happen often.

9.4.1.2 C -Customer requirements

The modules should support the different requirements of the various customers. No customer should have to buy more functionality than he wants. With few FABs in each module this condition can be met. If a few more FABs are included in each module it is still possible to meet this condition if the FABs are carefully selected so that they are logically connected. Using options also aids in meeting the requirements of the customer. At around six or seven FABs it is no longer possible to sell only the functionality that a customer wants but he also has to purchase functions he does not need.

9.4.1.3 P - Parallelism available in the development process

The amount of parallelism that is available in the development process is directly related to the number of modules in a system. Thus, if there are many FABs in each module there are few modules and so there is not much parallelism available.

9.4.1.4 K - Knowledge needed

The knowledge that an individual creating a module has to have increases as the number of FABs in each module increases. There is a limit to what is reasonable, but around ten FABs is considered the maximum. Today a complete system of about twenty FABs is the limit and thus about half of that seems like a reasonable amount.

9.4.1.5 E - Engineering

The engineering of a system gets more complicated as the number of modules in a system increases. It is therefore desirable to have as few modules as possible. About six or seven modules is about the maximum that can be handled when doing the engineering. If there are more modules it will be difficult to get the general overview of the system that is needed.

9.4.1.6 H - Handling

The cost of handling is directly related to the number of modules. The cost of writing documents, maintaining tapes and so on is almost constant for a module and not affected by the number of FABs in the module.

9.4.1.7 D - Dependencies

It is desirable to avoid dependencies between modules. All dependencies between FABs should be contained within the module. In XM it is sufficient to group three to four FABs together in each to entirely avoid dependencies between modules.

9.4.2 Licenses and options

The best way of handling options is to distribute them on the same tape as the rest of the module and then use licensing procedures.This will keep the cost of handling options down. All parameters are described, even those that belong to options not installed. When a customer wants to add an option the only thing that has to be done is to ‘unlock’ the option with a license code and the option will be operationable.

Pricing will also be simplified by using licensing procedures since handling modules will be less dependent on customer requirements. Today customers are forced to buy functionality that they do not want just because that functionality can not be ‘locked’. The question is how much Ericsson can charge for these functions. With licensing procedures these delicate questions could be avoided.

When creating a module the number of options in the module should be kept to a minimum. This will reduce complexity since complexity grows with the number of options. The need for a large number of options may indicate that the module is incorrectly designed. If there is a large number of options in a module and those options are not installed, there will be much unused capacity in the system since the requirements of a module do not change whether an option is installed or not.

Third party software should not be optional in a module. Even if a customer decides to leave the optional third party software out it still has to be installed, although locked, together with the rest of the module. This means that Ericsson still has to pay for the licence. Hardware should not be optional either. If an option requires special hardware and the option must be in a special module it must still be possible to install all software even if the hardware is not present. Unlocking that option may still be a little difficult. In addition to unlocking the option the hardware has to be installed and it is possible that the module needs to be somewhat reconfigured.

The best way to handle optional third party software and optional hardware is to put them in a module of their own and make them mandatory. This way most problems are avoided at the cost of an additional module.

Options in a module may not be dependent on other modules. Consider the following case. In module A there is an optional part that is dependent on module B. One customer does not want that option in module A, but in order to be able to install module A he still has to have module B. The customer is therefore forced to buy a module that he does not use. By not allowing options in modules to be dependent on other modules these problems are avoided.

9.4.3 Other things to consider when creating modules

The most vital thing to consider when creating modules is that the components in the module must be logically connected. This must have greater priority than the actual size of a module. There is no upper limit for the size of a module as long as it is still possible to handle the module and most customers require all the mandatory parts in the module, e.g. the platform.

It is important to avoid dependencies on other modules (other than the platform). Avoiding horizontal dependencies will give a hint as to which FABs are logically connected and will also reduce the amount of description that is needed for the module.

The dependencies that need to be described on the FAB level are dependencies between the different FABs in different modules. The dependencies are such that one FAB will not function or will function incorrectly if another FAB is not present in the system. This type of dependency is very rare, but if for some reason they need to exist there must be a way of describing them.

To simplify handling the modules are equipped with a default configuration that Design has worked out. This means that many modules can be installed without needing to configure it.

If many customers only want a part of the mandatory part of a module, that module may be split into two modules. The reason that a customer wants only a part of a module is often that there exists a logical gap in the module.

If a module is stable, i.e. if it is not revised often, and the module contains functions that the customer always purchases together, the module can be larger. An example would be the platform, which all customers need.

10 Advantages of modular thinking

One of the reasons for modular handling is that it makes it possible to handle each module independently of the other modules, allowing one person to be totally responsible for each module. This person can have full economic control of the module and therefore charge the right price for it. If the sales of the module drop this will be obvious to this person and action can be taken, e.g. better marketing, modifications or simply stop selling the module. There are several other benefits of modular handling, and these are described in the following sections.

10.1 Simplified engineering

- There is no need to search in many places to get hold of required information in most cases. Today, when information is found at the CXC level it is often too detailed. The time and knowledge needed to generate a customer specific system is therefore reduced.

- If there is a description of dependencies to other modules, combinations of different modules can be installed in a reliable manner. With a dependency description it is possible to foresee where problems may occur, e.g. two modules use the same database or two modules send drag & drop objects to each other.

- Today an AL is already engineered, enabling standard systems to be created fairly easily. It is when a system that can not be derived from an AL has to be created that a lot of work has to be done. Modules require a little more work every time a system is to be created, but this is outweighed by the fact that there are no special cases that require large amounts of work.

- A well defined module makes configuration much easier. If one or more modules are added to an existing system it is easier to rescale and reconfigure the system, since every module has its own well specified requirements on the system. There is no need to guess or test.

10.2 Simplified handling

- The need to rewrite engineering data documents for specially advanced customers is reduced, since no extra FABs which can alter the engineering data can be

included in a module. If new hardware configurations are used it might of course be necessary to rewrite the engineering data for the module.

- Using licensing procedures the pricing of the system will be less connected with the handling of the system. This will make handling price profiles easier since the handling needs to be considered less. The problem with charging for functionality that the customer does not need is avoided.

- The flexibility of the system will increase, since the system can be customized according to the customer's needs.

- It is possible to decentralize knowledge so that a few people are experts on one module each. This is not possible today because one has to know a lot about all FABs to be able to understand how they are related.

- It is also easier to get an overview of the system when it consists of several well defined 'black boxes'. At least if the 'black boxes' consist of logically connected functions.

- A revision of an application, i.e. error correction, can be done on a separate module instead of as today on a whole AL. Each module can live its own life independent of the other modules. Some modules change frequently while others seldom change. That way the time from an error report to the correction of the error on site is reduced because it is only the module that requires a new revision, not the whole AL.

- Modular handling aids in structuring TMOS. By requiring that modules be logically connected and by enforcing a way of describing interfaces odd dependencies will be detected and it will then possible to change the design so that these dependencies are eliminated.

10.3 Simplified testing

- The need to make comprehensive tests when adding new functionality to an existing system is reduced since it is sufficient to test only the parts that are dependent on the new module that contains the new functionality.

11 Problems with modular thinking

There are some drawbacks in using modules in the TMOS system if no extensions are made to how modules are used. The problems are:

- The first installation of a system, i.e. maiden installation, has to be done in the same manner as upgrades of the system. First by installing the platform and then adding one module at the time. The reason for this is that no superior documents exist describing how to handle a complete system.

- In some cases it will probably be necessary to have access to all module description documents for the previously installed modules, as some of the load estimations of resources may need parameters from all installed modules that use the resource. It is suggested to make the configuration formulas and guidelines so that parameters from other modules are not needed, but this is probably hard to do. Since every module has its own set of documents which sometimes have to be used, it will probably be a small problem to handle them especially at a maiden installation.

- Another problem is that the module description document is supposed to be written by the Design departments, who do not have a good overview of the system since all FABs are not developed in the same place. It would be necessary to have a person who is responsible for each module and who has a wider knowledge of the system. He should arrange for all the different parts of the module description document to be written by the people who know the subjects best and then put them together into a description document of the module, i.e. an extended engineering data document.

- A great many man hours need to be spent on developing all the formulas and guidelines for estimating the load on resources. Today very few resources have this kind of description. It is probably very hard to calculate the load on a resource especially if the results should be in terms of delta values, i.e. how much a parameter has to be increased, and not an absolute value. This is in order to be able to ignore all previously installed modules that use the same resource.

- One problem is that models are needed for estimating load on and performance of resources. If calculation shows that a resource is adequate there is no need to test that it is. These models are used instead of tests. The problem occurs if the models are erroneous. This could lead to errors that would have been found with tests.

- One of the goals when introducing the module concept into the TMOS system was to eliminate the need for functionality tests when adding a new module to a system. To be 100% sure that one module does not affect another, every little dependency has to be thoroughly described and every available resource has to be defined. The amount of work this would demand is not in proportion to the gains. It is therefore necessary to choose which resources to describe. This introduces problems since it is difficult to create the rules for which resources that have to be described. This in turn opens up the possibility of making an error when describing resources by accidentally leaving out important information. The result is that it is not possible to be 100% sure that modules will work together, but one can be sure enough if the important dependencies are described.

Chapter 13 describes some ways of easing these drawbacks.

12 Describing dependencies

In order to know which parts of a system are affected by a module there has to be a way of describing how a module affects its environment. This is in order to know if the module will work in the environment and if the module uses some resources a lot. These commonly used resources might have to be resized to increase the capacity of the resource.

A dependency between modules is introduced when one module has a resource that another module wants to use. Since the concept of resources will be used in the following discussion a basic definition is given and then details in the definition of resources will be discussed. A generic resource is shown in figure 10.

12.1 What is a resource?

- A resource is something in one module that another module might need. A resource can be hardware, such as disk, communication channels and memory or intangible things such as port numbers. Databases and services provided by servers are also resources. Hardware resources belong to the platform module. In the description of a module there is a part describing the resources it uses:

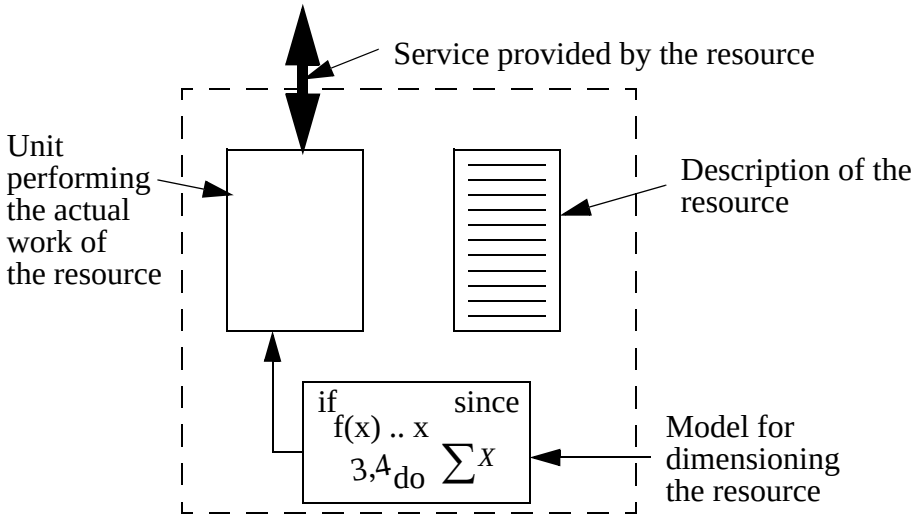

Figure 10. Generic resource

- A description is associated with each resource. This describes what the resource is and a model for dimensioning the resource is also included.

- All problems with interconnecting modules are related to resources. Since a resource is something in one module that another module needs, all interconnections between modules can be looked upon as a module needing resources in other modules. Then, if all interconnections between modules are related to resources all problems with the interconnections are also related to resources. Since two objects that do not share anything can not affect each other no conflicts can arise. It is when two objects share something that a conflict may arise. If that something that they share is defined as a resource, that conflict may be detected and also solved.

12.2 Size of resources

One problem is deciding the size of a resource, and a problem which is related is what a resource is. A few examples might help to clarify what size is. For example, the information model is a large resource, while a port number is a small resource. Size is not to be confused with the performance of a resource. The size of a resource has to be a trade-off between the exactness of the description and the amount of documentation to describe all resources in a TMOS system. If the resources are large, there will be few resources in a system and thus the amount of documentation will be reduced, although the exactness of the description will also be reduced. Fortunately, the size of the various resources can vary in a system such that a better compromise between the size an exactness can be achieved. Those resources that need exactness in the description can be small and the other resources can be larger, thus reducing the amount of documentation. The description of dependencies is intended to detect and resolve all conflicts between modules and because of this the following can be concluded about the size of resources: ‘The size of a resource should be such that it is as large as possible while being exact enough to resolve all conflicts that may arise.’ Anticipating all conflicts that may arise is not easy and it can not be done completely, but by looking at known problems and following predefined guidelines a relatively good approximation can be made. It is important to note that although the resource concept may be difficult to handle, in theory it provides a complete method for detecting all conflicts.

12.3 Which resources shall be described?

If all resources are described there will be a great deal of documentation. It is therefore important to reduce the amount of documentation in order to reduce cost. It is important to note, however, that if all resources are not described all possible conflicts will not be detected. But if the description of the resources that are related to highly unlikely conflicts are left out, a good trade-off between cost and completeness can be achieved. However, conflicts that result from some resources not being documented can be hard to track down.

Another thing to note is that when the amount of documentation is reduced, the rules for how to document things become more complex. It is relatively easy to have a rule requiring the documentation of everything. When only some things have to be documented, rules for what shall be documented have to be added. It is important that the rules for describing dependencies be clear and unambiguous. If they are not, there will be misunderstandings that will result in errors. Unfortunately, there is no known way of stating such rules, but it is nonetheless important to try to be as clear as possible.

One important observation that can be made is that it is only when a resource is scarce for some reason that the quantity has to be known. It is for instance interesting to know that a database uses 20 Mb on a device if the device can only hold 100 Mb. It is not interesting to know how much space that database uses if the device has an infinite amount of storage. Of course, nothing in a computer is infinite but some resources are almost limitless while others present real limits. The resources that are limiting are called bottlenecks. The important thing to note is that it is only dependencies on bottlenecks that need to be described with quantity. Of course, determining which resources are bottlenecks might present a serious problem. It may be hard for an assistant programmer to determine which resources really are bottlenecks. It will probably require experienced personnel and measurements to find the bottlenecks. Fortunately, if a few extra resources are added for safety's sake, the only penalty will be additional documentation.

Even if it is not necessary to describe quantities for resources that are not scarce the resources still have to be there. This would call for a description of all resources that a module uses. But there are many resources that can be thought of as obvious, such as network elements, communication boards (HSI board) and PMS and those obvious resources can be left out. Note that resources that are bottlenecks still need to be described even if they are obvious. Determining what is an obvious resource may be really difficult since what is obvious to one person may not be obvious to another. One way of solving this problem may be to create a list of all the obvious resources that exist e.g. network elements, PMS and so on. If such a list is not created people will have to decide for themselves what is obvious and this will lead to errors. One way to define the obvious resources is to say that all resources that the platform provides are obvious.

12.4 Where shall resources be described?

Another problem is related to where the description of the resource shall be kept. There are two different approaches that can be used. The information can be put in the module that uses a resource or in the module that provides the resource.

Keeping the information together with the module that uses the resource results in a minimum of describing, since only those resources that are actually used are described. Although this might seem like a good approach at first, this method

has several drawbacks. There will inevitably be multiple descriptions of the same resource scattered in the various modules. This will create problems with keeping the descriptions consistent and since the same description is found in many places the amount of documentation will grow.

The other way is to keep the description of a resource together with the module that provides the resource. The drawback here is that there is a chance that resources that are not used are described anyway and this will create more documentation than necessary. Most resources are defined because they are needed so this might not be a big problem. The advantage with this way of describing is that each resource is only described once and thus the problems with keeping multiple descriptions consistent disappears. This is also consistent with the object-oriented thinking, a way of reasoning that has been widely accepted.

12.5 A model for describing resources

There are many ways of combining these demands into a method of describing dependencies among modules. One combination is given here and that combination is discussed in more detail. Other combinations are of course possible, but this is believed to offer the best trade-off between cost, ease of handling and the number of conflicts that can be successfully detected.

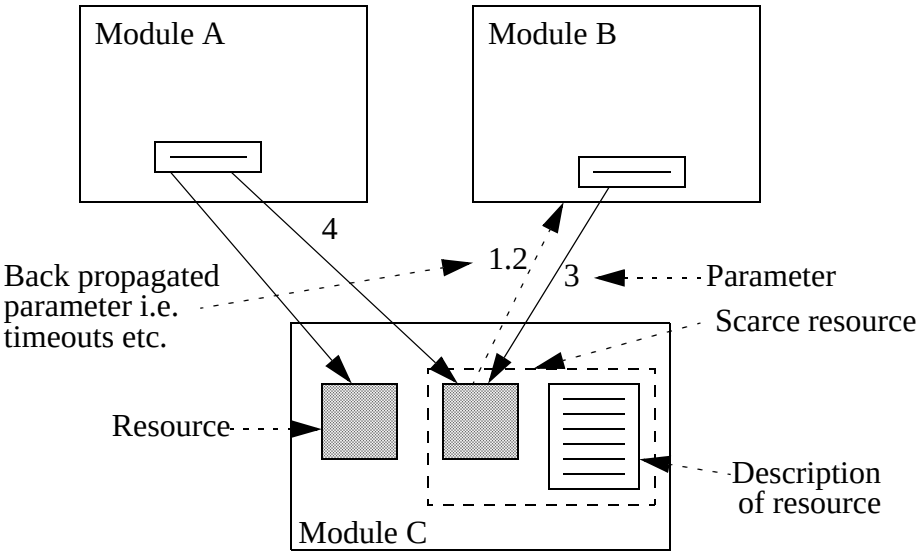

Figure 11. Model for describing dependencies

The resources that are scarce, or might present a bottleneck, are described in the module that provides the resource, module C in figure 11. In this description a

model for dimensioning the resource is included. Input parameters to this model are provided by the modules needing the resource, modules A and B.

In the description of a how to dimension a resource there is also a description on how to expand the resource if more performance from that resource is needed. The reconfiguration of a resource must be very simple. In most cases it should be enough to change only one parameter to reconfigure a resource.

Dependencies on resources that are not bottlenecks are described in a simplified form. This information is used when testing the module in the new environment. In the modules that use a resource (module A) the fact that they do so is stated. This information is needed in order to determine which modules are needed in a system. There is no description in the module that provides the resource (module C). This way it is possible to handle the use of such things that the creator of the module did not anticipate from the beginning.

Dependencies on obvious resources are not documented anywhere, which considerably reduces the amount of documentation. Obvious resources that are bottlenecks and dependencies on those resources still need to be described.

It is possible that some parameters, e.g. time-out values from a resource need to be back propagated to a module, the parameter from module C to B in figure 11. This should, however, be avoided whenever possible.

In some cases, a resource is accessed via another resource. This is the case when modules subscribe to events. The event is sent to the platform by one module and then distributed to all the subscribing modules. In these cases it is of course not the distribution-process in the platform that is the actual resource (if the process is not a bottleneck). Information of how many events, orders or messages that are sent to the platform is to be described. If it is possible to subscribe on the information, names of the subscriptions should be included. In this way it is possible to match the subscriptions in the dependency description.

All modules are equipped with a standard configuration so that there are only a few parameters that need to be changed when the module is to be installed.

12.6 Pros and cons with describing resources

The benefit of this approach is that the amount of documentation is kept to a minimum. The only extra (not useful) descriptions that might occur are those describing resources that are not used by anyone. This may not be a serious problem since most resources are created because they are needed by someone.

By describing the dependencies it is also possible to determine which modules or parts of modules that need to be tested when changes are made to a module.

There will be no need to test the whole system just because one module has changed.

It will be difficult to keep track of all the resources since a resource can be defined by using it. Although this is a drawback, this is outweighed by the fact that there is no need to administer all resources in the TMOS system. It does not affect the model if all resources are defined and registered centrally.

12.6.1 Describing performance with formulas

To be able to know if a resource is to be reconfigured it is necessary to predict the load on and capacity of the resource. This is one of the big problems with modular handling. Is is perhaps possible to make some kind of computer related constants that can be used in formulas. If there is no such constant, it is hard to estimate how much of a resource's capacity a dependant module is using. Another way is to give examples of different hardware configurations and how the resource's capacity relates to them.

12.6.2 Problems when adding a new module

One problem arises when a new module is to be added to a system. This is illustrated in figure 12. The system consists of the platform, module A and module B. Module C is to be added.

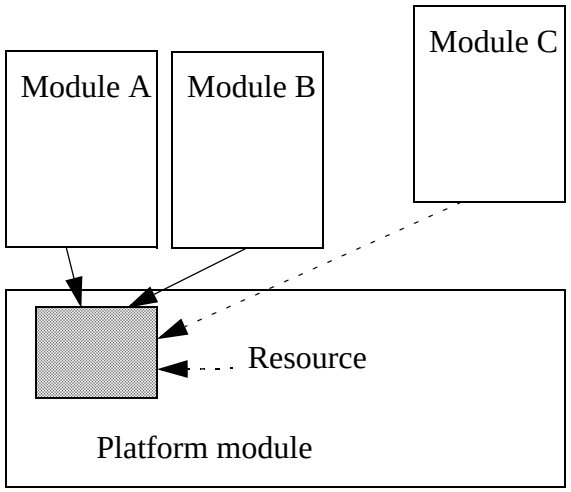

Figure 12. Adding a new module to a system

The problem occurs if module C uses the same resource as A and B. In this case the document describing module A, B, and C has to be gone through to find the parameters that are needed to dimension the resource in the platform. This is an unwanted effect and there are two things that can be done to get around it.

The first thing is to have a procedure to determine how much of the resource is already used. For example, to determine how much space that is used on a disk UNIX is “asked”. Such procedures can be created for all resources eliminating the need to check all documents.

The other thing that can be done is to document how much of a resource is used. This documentation can be created either manually with “pen and paper” or automatically by some kind of resource information database in the system.

Both of these solutions require that it is possible to redimension the resource only by knowing how much is already used and the parameters from the new module. If none of these two approaches are implemented or if the results need to be back propagated the only thing left is to go through all the documentation. In order to ease this task the documentation has to be as standardized as possible so that the same information can be found at the same place in all documents. The number of parameters that has to be changed when adding a new module must be kept to a minimum, e.g. changing the size of a database should be done by changing a number in only one place in the system.

13 Additional simplification

The concept with modules and resources is sufficiently powerful to allow handling of TMOS in the future, but handling might still be a little cumbersome. Two additions are therefore made that simplify handling even more.

13.1 Predefined platform

Even if the platform is like any other module it still has a few characteristics that make it unique. These characteristics can be used to simplify the handling of the system.

- Most resources are located in the platform.

- All modules are dependent on the platform. There will thus be a platform in every system.

- Many resources in the platform are interfaces. Another program is often needed to make use of that resource.

13.1.1 Using the platform to define obvious resources

Since the platform is present in every system and many resources are located in the platform it is possible to let the platform define the obvious resources, considerably reducing the documentation describing dependencies.

Dependencies on options in the platform must still be described since there is no guarantee that the option will be present. It is therefore vital to keep the number of options in the platform as small as possible.

Many options need another program to take advantage that option, e.g. interfaces. If the option is made mandatory in the platform this does not matter. The program using the option can be sold at a higher price.

13.1.2 Preconfigured platform

The platform is special since all other modules depend on it and thus the platform is present in every system. By having a preconfigured platform defined for each THP the installation can be greatly simplified.

If the installation were to be done strictly abiding by the modular concept, there would be many unnecessary steps. In order to illustrate this, suppose that a sys-

tem with three modules A, B, and C, and a platform are going to be installed, see figure 13. The following steps would then have to be performed:

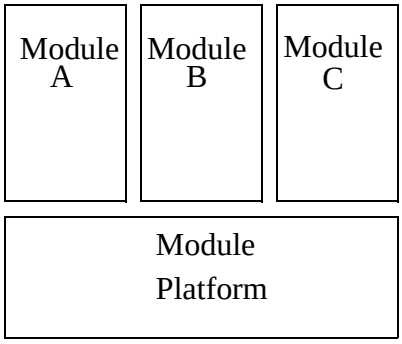

Figure 13. An example system

1. Install the platform module. Note that all bottlenecks in the platform do not have any capacity now.
2. Use the documentation for module A to see which dependencies that module has on the environment, and calculate the new configuration for the resources that the module uses.
3. Reconfigure the resources in the platform to accommodate the new module.
4. Install module A.
5. Use the documentation for module B (and possibly module A) to find out which resources that module B uses and recalculate those resources.
6. Reconfigure the resources that module B uses.
7. Install module B
8. Repeat steps 5 through 7 for module C.

One way of easing the procedure is to have the platform preconfigured, i.e. many of the resources are already dimensioned so that the resources in the platform can accommodate a number of modules before the resources need to be reconfigured.

Another simplification that can be done is to do all the calculations before any modules are installed. That way one can do all the calculations at home and then go to a site to install all the modules. The installation procedure would now be as follows:

1. Use the documentation for modules A, B, and C to find out which dependencies they have on the platform and on each other. Calculate the parameters for the resources both in the platform and in the modules.

2. Check if the resources in the platform can accommodate the modules. If the resources in the preconfigured platform are dimensioned properly the platform will accommodate most combinations of modules.
3. Install the platform. In most cases no configuration will have to be done. If the modules did not fit, the platform must be reconfigured.
4. Install modules A, B and C. Note that special dependencies between modules may prevent the modules from being able to be installed in parallel.
5. If the modules were dependent on each other configure the resources in the modules.

13.2 Ready-made calculations

With a preconfigured platform the step of reconfiguring the resources in the platform does not have to be performed in most cases, but there is still a need to do all the calculations in order to determine if it is safe to install the modules. But this step can be circumvented in many cases too. Figure 14 illustrates the normal actions to create a system.

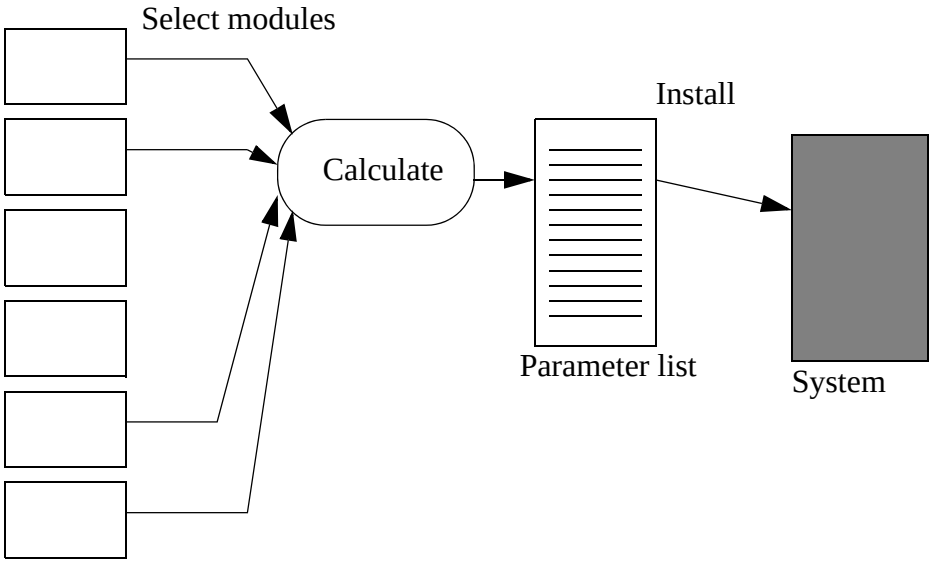

Figure 14. Procedure to create a system

If the parameter list is kept the next time the same combination of modules is to be installed, the same parameter list can be used again. This is similar to the AL concept that is used today, but with one big difference, namely that it is still possible to determine if it is possible to add another module to such a system and how adding a module is done.

All documentation is still at module level. The only documentation that has to be present on the new AL level is the parameter list, a description of which combination of modules and THP that list belongs to.

Of course it would be necessary to redo some or possibly even all calculations when a module is revised, but if the modules do not change frequently it will be possible to reuse the calculations to quite a high degree.

Another problem is that many combinations of modules exist together with different options in modules. But since all modules are configured as if all options were present, the number of combinations will not be so large. For the same reason there will be some flexibility allowed within each AL.

This new AL does not allow customizing of applications in any way. If a customer wants a larger or a smaller system, e.g. database sizes, a new AL has to be created. This AL can then be used by other customers that have similar needs.

13.3 Preconfigured modules

When modules are to be installed there are a number of parameters that have to be set, but many parameters are usually the same. If the modules had a standard configuration there would only be a need to change a few parameters when the module is installed.

Parameters that are internal in the module and therefore not needed outside of the module, are set by the creator of the module. These parameters can not be changed at all, because there is no need to change them. This eliminates the errors caused by accidentally changing parameters that should not have been changed.

13.4 Resource server

A second step in simplifying the handling of the system might be to create an automatic tool that keeps track of all the resources in a system. The modules or the system administrator can use the tool to see if there are enough resources available in the system if another module is to be installed. When a module is installed it will tell this tool (resource server) how many resources it uses and the resource server will then update its data.

This tool could possibly be used standalone as a tool for the people doing engineering and installation. It could also be used by sales staff who would be able to tell the customer in a few seconds if he can install another module in his system or if he has to upgrade his hardware also.

14 Demands and restrictions when using modules

To be able to use the modules efficiently and successfully, there are some demands that must always be met and some that should be met if possible.

It is important that everyone follows these demands and that the modules are created so that they adhere to the standard. This is to make the modules work as efficiently as possible. To make it possible to follow the demands, well defined design rules are needed. The demands that the modular concept places on the design of modules are presented here.

It is also important that the engineering does not have to be done for each new system. It is therefore important to create the parameter lists and then derive the systems from those lists.

14.1 Modules

- It must be possible to reconfigure the resources in a module. This requires that a parameter is only defined in only one place. If this rule is not followed, it will inevitably occur that a parameter is changed in one place but not another, which will lead to strange errors that are hard to find.

- A module should not be dependent on other modules except the platform. Modules should especially not be mutually dependent on each other. Handling will be greatly simplified if modules are dependent on the platform only.

- It must be possible to locate bottlenecks. This is necessary to determine which resources must be described completely and which resources only need a simplified description. This reduces the amount of documentation.

- Resources must not be overdimensioned, i.e. no safety margin must be added to the demands on the resources. Adding safety margins can be done when a system is created.

- All parameters must have default values, which reduces the number of parameters that need to be changed when the module is installed.

- One module may read data from or add data to another module’s database. This should preferably be done with the server-processes that administrate the data-

bases. But modules may not alter or delete data, or add or delete tables or relations. If a module has to alter or delete data in another modules database, this must be done by using an interface in the module that owns the database, not through direct manipulation of the database.

- One module may read files in another module, but it may not alter them in any way.

- Back propagated parameters should be avoided, enabling the amount of work that has to be done when a system is engineered to be considerable reduced.

- One CXC may not appear in more than one module.

- Installation dependencies must be avoided, i.e. one module has to be installed before another. Installation dependencies make it harder to administer installation and makes error recovery of the installation more complex, i.e. if the installation of a module fails, errors can also occur in other modules.

- There should be one person responsible for each module. This person can have complete control over a module, making it easy to see if each module is generating profit.

14.2 Options

- If an option in a module is dependent on hardware, it must still be possible to install that module even if the hardware is not present. An example would be a communication module that requires special I/O hardware to work. It must be possible to install that module even if the hardware is not present. This way the customer is not forced to buy hardware that he does not use. In many cases it is better to place the hardware and software in a separate module.

- If an option is dependent on hardware it must be possible to reconfigure the module to accommodate the new hardware when the option is unlocked.

- An option in a module must not depend on another module, this way the customer is not forced to buy modules that he does not need. If an option in a module depends on another module, that option should be moved to the module it depends on. If modules are created from logically connected functions this problem should not occur.

- Licensing procedures must exist. If not, it will not be possible to handle options in a module efficiently and the whole module concept will collapse.

- Third party products should not be optional. If a third party product is optional and the customer leaves that option out, the software still has to be purchased by Ericsson and installed, even if it is locked.

14.3 Platform

- The platform should not have options. This reduces handling and the amount of documentation in a system. In certain cases options may be necessary though.

- The platform should be preconfigured so that it fits different combinations of modules without changes. Installation of new modules will then be simplified, since it will not be necessary to alter the resources in the platform every time a new module is to be installed.

- The platform should be able to handle all TMOS applications. All applications should be built on the same platform. Using only one platform will reduce problems when two TMOS applications are to be installed in the same system.

- There must exist a way of determining how much capacity there is left in a resource.

14.4 Installation

- It must be possible to define all parameters before the installation of a module. When the module is to be installed all parameters are entered and then the whole module will be installed without user interaction.

- No workarounds are allowed in the installation procedure. This will reduce the time it takes to do the installation by reducing the amount of user action and will also reduce errors due to typing errors from the operator.

- Modules must be completely installed by their own installation script. All authority and configuration of user categories, e.g. .login files should be handled by the module.

- It must be possible for a person to install a module without detailed knowledge of the whole system.

14.5 Documentation

- A list of all obvious resources must exist. It is basically a list of all the resources in the platform.

- The documents describing the module must follow the standard closely so that it is easy to find the required information quickly.

- Documentation of how to rescale and modify a system has to be written in order to be able to add functionality. This could be how to expand form one THP to a larger one, or adding harddisks or describing complex resource reconfigurations. These documents are referred to in the module description document.

15 Comparison of modules and AL

There are many aspects that need to be considered when comparing modules and AL. In some aspects the two are similar while in others they differ.

15.1 Adding functionality to an existing system

Adding functionality to an AL-based system is easy if that functionality is part of the AL, i.e. options. But adding functionality that is not part of the AL can be very difficult in most cases. This is due to the fact that there is no way of knowing which resources are needed and how the new functionality will affect the system.

Adding functionality to a system based on modules is easier. If an option is to be added to a module all that has to be done is to unlock that option. Adding another module to the system is a little more complicated, but it can still be done reliably with a limited amount of work.

15.2 Scaling of systems

Scaling of systems is difficult in both cases if scaling requires the addition of hardware. This is because it is difficult to follow the THPs that exist especially if the existing THP is a few years old. Consider a system that was new two years ago and consisted of two SUN SS10. It is not possible today to buy two new SS10 in order to upgrade to the next THP.

The advantage of modules is that there are scaling procedures included in the module description document.

15.3 Configurability

It is possible to configure an AL so that it fits more different needs within the AL than with modules. Module based systems can not be adapted to such a great variety of customers within an AS, but it is easier to create a new module-based system than it is to create a new AL. The reason for this is that the modular concept is intended to simplify handling and therefore the modules are more standardized and some configurability is omitted.

15.4 Installation

Installation time does not vary much between modules and AL. The factor that influences installation time the most is how the installation procedure is designed. To reduce the installation time as much as possible all parameters should be entered beforehand in some kind of database and then the rest of the installation should be automatic with no operator interaction.

16 A module description document

The purpose of a module description document is to make modular handling possible. The document can be looked upon as an extended engineering data document. The information that differs from the engineering data document is descriptions of dependencies to other modules and what these dependencies consist of, a description of the available resources that the module offers to other modules, and a description of how to reconfigure the system if a module has requirements a resource that can not be met by the present configuration. It is important to state that all the information in this document should be on a need-to-know basis, i.e. no unnecessary information should be included. If a special configuration problem occurs it has to be dealt with at CXC description level or, if that is not possible either, by asking the designer. The descriptions at CXC level do not have to describe dependencies since the amount of work needed to create that many dependency descriptions at this level is very great. The module description document does not include details at CXC level in order to simplify the ordinary handling of modules.

The important headings in the document are:

- Short description
- Table of dependencies
- Dependencies
- Engineering data
- Available critical resources

16.1 How to use the document

The document can be used in several different situations e.g. a maiden installation or when adding a module to an existing system, but they all end up in almost the same usage procedure. Some different usage procedures will be illustrated here.

16.1.1 Creating a system specific parameter list

A system specific parameter list is a pre-engineered list of all parameters in a system. A system is a combination of different modules. The reason for keeping a parameter list for each specific system is that the configuration of a system takes time. If the configuration of a site can be done by using an old system configuration a lot of time can be saved. The probability that a similar system has been

configured before is quite good, since only the modules need to be the same and not the options within them.

The documents needed for the configuration of a new system are the module description documents and parameter lists for all the modules that are to be configured. A system specific parameter list is created as follows:

1. Check if all dependencies can be handled by the modules that are to be installed or if other modules or options have to be added as well. This is done by using the ‘table of dependencies’. This table also tells which modules must be installed before others.
2. Find out if there are any conflicts in dependencies between the modules. This is done by looking in all module description documents to see if they are dependent on the same resources or if they have other dependencies that could cause problems, e.g. different communication protocols. All these dependencies are described in the section ‘Dependencies’. Then check if conflicts may occur; the parameters that are to be set due to requirements of dependencies must be written down in a new system specific parameter list document that is needed at installation time. If modules use the same resources the parameters must be summed and calculated according to instructions in the section ‘Available critical resources’ in the document for the module which contains the resource (in most cases the platform-module). In some rare cases parameters can be sent back from the calculation to the dependent module. These parameters must also be included in the parameter list.
3. Now all dependent parameters are set so that no conflicts are caused and all requirements of the modules are met. This work has be done in a parallel manner, i.e. all documents must be considered at the same time in order to sum all the modules’ requirements. The rest of the configuration can be done sequentially, one module at the time.
4. The parameters in the parameter list for the modules are transferred to the system specific parameter list. Note that only system specific parameters have so far been set and site specific parameters, e.g. hostnames and ip-addresses are not yet set.

16.1.2 Configuring and installing a system

In this section it is assumed that a system specific parameter list is available for the kind of system that is to be installed. If there is no such parameter list, one should be made according to the section above. All maiden installation documents, for the modules, should also be available. The module description documents are needed for naming hosts and servers for instance, according to given standards. Configuration and installation are done as follows.

1. The system specific parameter list should now be altered so that it fits the specific site where the installation is to be done. This is done by exchanging all variable names with site specific names and figures. The new parameter list can not be used by any other sites but the one it is created for.
2. The site specific parameter list can now be used together with the maiden installation documents for each module. The installation has to be done in the order that was concluded in 14.1.1.

16.1.3 Adding a new module to an existing system

When adding a new module to a system it is vital to be able to reconfigure the system according to the new module's requirements on the system and knowing that the module does not influence the functionality of the existing system. The documents needed to perform the installation are the module description document, the maiden installation document and the parameter list document for the module that is to be installed. Furthermore, all module description documents for the installed modules are needed. The procedure for adding a new module is not very complicated.

1. Check if the module needs any other modules than those already installed in the system. If that is the case these must be installed as well. If the module is dependent on an option in an existing module and this option is not licensed, this option has to be 'unlocked'. These dependencies are described in the section 'Table of dependencies' in the module description document.
2. Some of the dependency in the section 'Dependencies' involves redimensioning different resources. In order to do so one needs to look up which of the other modules is dependent on the resource, sum the requirements and calculate the dimensioning parameters. The demands on the resources are described in the 'Dependencies' section and the resources and formulas are described in the 'Available resources' section together with a description of how to find out how much of the resource t is currently used. If the resource that is to be redimensioned can not match the requirements there is a description or a reference to another document in the 'Available resources' section on how to increase the performance of the resource. This procedure is done for every dependency that the module has.
3. The system is now reconfigured so that it can match the requirements of the new module. The maiden installation is done as stated by the maiden installation document and the parameter list.

16.1.4 How to use the document when testing a module

In order to test if a module is correctly installed it is necessary to be able to determine which functions are important to test and which functions in the system are not affected by the new module. This is done by looking in the ‘Dependencies’ section of the module description document where all the dependencies of the modules are described. Since a dependency can affect a resource that is used by another module it is necessary to look in all the module description documents to see which other modules are dependent on the resource. The functions in the module that uses the resource must be tested. The descriptions of the dependencies can indicate if there is a need to test the relation. When all dependencies are traced and tested, a minimal yet adequate test has been done.

17 How to write a module description document

In order to ease creation and standardization of the document a number of headings are provided that must be filled with information. In a separate document a fictitious module description document is presented to give a more concrete understanding of what a document will look like.

17.1 Short description

A short description of the functionality of the module, what it does and which services it provides. This description must be kept very short and concise. All mandatory FABs can be described in one section but the optional FABs should be described one by one.

A table of which FABs the module contains and if they are optional or mandatory is placed in this section. References to vital documentation for the FABs could be included in the table.

This short description will also give beginners an introduction, thus enhancing understanding of the system.

17.2 Table of dependencies

This is a description of which other modules must be present in the system in order for the module to function correctly. These dependencies should also be available in PRIM but it will not be possible to identify which FABs in a module need to be active. In order to be sure that a module will work correctly, not only the modules to which it has dependencies must exist, but the FABs in the modules must be chosen if it is an optional FAB. A dependency table can be made with a short description of the dependencies.

FAB	Dependant of	In module	Dep.	Description
FAB_760_000	FAB_760_001	modul_1	I,R	uses same database
FAB_760_002	FAB_760_003 or FAB_760_004 and FAB_760_005	modul_2 or modul_3 and module_4	I I,R R	uses their data to calculate results
FAB_760_003	FAB_760_006	module_5	I,R	sends messages

Figure 15. Sample table of dependencies

The dependencies, (Dep.) in figure 15, indicate if there is an installation dependency an that module (I= Installation dependency, R=Runtime dependency), i.e. if the module has to be installed prior to the installation of the module. If there is an I in the Dep. column there exists an installation dependency, exists.

Note that dependencies between modules are very rare. Most modules depend only on the platform. The table of dependencies will therefore be almost empty for most modules.

17.3 Dependencies

In this section in the document dependencies are to be described. It is important to point out that dependencies between modules are very rare, but dependencies exist to other modules indirectly, since modules share resources in the platform. This fact does not affect the way dependencies are described (see chapter 12)

There are two different kinds of dependencies. First there is the dependency that does not require any (almost) performance from the resource. This is by far the most common kind of dependency. And second there is the dependency that requires, for instance, database size, CPU power or communication links. The difference is merely that in the second case one has to describe how the resource is used in term of load, usage and other requirements while in the first case it is only relevant to know that the dependency exists, so that the module can be included in the system without causing undocumented side effects. In order to avoid the problem with dependencies to optional FABs in modules it is necessary to specify in which FAB the requested resource is contained.

It is important to be able to find specific dependencies in an easy way, especially when looking for which modules are dependant of a certain resource. This is

done when reconfiguring a resource or when a new module is to be tested. To simplify the search procedure the dependencies are divided into different categories e.g. Databases, Services and External communication. In each category the dependencies should be clearly separated with a short heading with the name of the resource.

17.3.1 Databases

In this section a description of all the databases that the module uses is made. A description of what the module does to the database is needed for determining whether combination of modules is ok, or if the modules affect the database in a conflicting way. This could for example be if a module changes tables or relations in the database or if it simply reads or writes data. If the module adds data to the database it may have to be resized, and formulas for this kind of reconfiguration must be described. The formulas will have some in-parameters such as number of alarms per hour, number of operators or how many NE are installed. The out-parameter should be in terms of how many additional Mbytes the database needs. The out-parameters are then used by the formulas in the resource description to the other module. If the module puts a high load on the database this should be presented in figures, such as requests per hour in different situations and average time for each request. If the static and dynamic requirements on storage for the databases differs this must also be described.

A description of when and why the module uses the individual databases. This information is included in order to ease the task of spreading the various databases among various servers.

17.3.2 Services

In this section all services that the module uses are described. Services could for example be fma, imh or communication interfaces. A service could be a request to another application in order to get information. It could also be a way of giving other applications a status of some kind. If a request puts a heavy load on the service server, information on the request rate and time for each request should be presented. This information could then be used to determine if a service server has to be reconfigured. Instructions of when and how reconfigurations are made are described in the section ‘available critical resources’ in the other module’s module description document. Dependencies are not to be described at process level. The description should be made at a higher abstract level (see section 12.2).

17.3.3 External communications

If the module communicates with the outside world a description of which protocols and what type of hardware is used is necessary. The configuration of external communication is often complicated, so a special description of these is needed. A description on the influence the module may have on other modules

and their external communication is also needed. This is important information since some communication protocols can not be installed on the same machine. Special configuration may be necessary if two different protocols are used in the same system.

17.3.4 Storage requirements

In order to dimension primary and secondary memory an estimate of how much of each type of memory is used is necessary. The estimates should be made at module level and typical usage of the module should be considered. If only parts of a module can be run simultaneously it can not use as much memory as if all parts are run. Many instances of programs in a module can be run at the same time and therefore require a lot of memory. Some parts are run rarely and others are run often. An estimation of storage requirements is clearly not easy to make, but some kind of guidelines are needed.

17.3.5 Other performance requirements

If possible the load on other resources such as CPU, network and so on must be described. Since performance is difficult to estimate and depends on many factors, examples from a few typical usages of the module can be included to give a rough description of the performance load.

If more detailed performance models are available parameters for them and references to the models must of course be included.

17.3.6 Other dependencies

It is possible that one module reads or writes in a file of another module. This should be described here at an high abstraction level i.e. not why. An example of this would be that modules add entries in the loginfiles of the various user profiles and these entries can conflict with other entries.

17.4 Engineering data

In this section a description on all the parameters that a module needs in order to be installed is included. This does not include the parameters that are predefined by Design (in the parameter list). With this information it is possible to define all parameters beforehand and then let the installation script take care of the complete installation of the module (see section 16). After starting installation no more parameters should be set and no files must be edited. This is probably not possible in reality, but it is a goal to aim for. The contents of this section is actually the same as in today’s engineering data document. The ‘engineering data’ section should roughly contain the following sections:

- Hardware configuration
  A description of how the hardware that belongs to the module is to be configured.

- Unix configuration
  If the module requires special Unix configuration, this should be described here.

- Patches
  Reference to a patch list.

- Module software
  A description of which FABs are included in the module, how to configure them and other information about them.

- Process summary
  A list of all the processes that are added to the system by the module.

17.5 Available critical resources

In this section a description of common critical resources that are available for other modules to use, is given. If a module wants to use a server-process in a module and this resource is critical, i.e. if the load on it severely affects its performance, the resource should be described here. Information on the capacity of the resource, and if it is possible to create more instances or reconfigure the resource should also be included. The description has to be limited to a minimum and only indicate a rough estimation of how the resource is affected by other modules. If it is possible to reconfigure the system to improve performance of the resource, a description of how to do this should be included here. If the reconfiguration is very complex, e.g. a hard-disk has to be added or if one has to upgrade the system to a two-server configuration, references to further reading are necessary. It is vital that only the resources that are bottlenecks, important or commonly used in the system are described, otherwise the list of possible resources will be endless (and only a few are actually used). A critical resource may be server-processes, databases or hardware.

17.5.1 Databases

Formulas for dimensioning databases according to the parameters given by the dependent module are described here. The database size parameters given by the dependant modules should only be summarized and increased by a safety-factor. A procedure for determining how much space that is available in a database should be described or a document describing such procedure should be given as a reference. If the available space is not enough there is a description of how to

resize a database, or references to other documents describing it should be included.

If the parameters given by the dependent module indicates that the load on the database is too high there is a description of how the databases can be divided to increase performance. How the distribution can be done must also be described, i.e. splitting the database onto different servers or different disks. Since it is hard to estimate when the load is too high only guidelines can be given, perhaps by means of examples.

A description of special demands that a database may have, for instance if a database requires truncating of log at installation or if other special configurations in the database server are needed.

17.5.2 Services

When describing an available service-resource this should be done at a high abstract level and not at process level. In some cases the module that uses the service describes the load with some parameters. Formulas for estimating the total load on the service should be included here. These formulas are probably quite hard to design and get relevant results from, but they can give an indication of the load. In some cases it is sufficient to give a maximum value of how the module can be loaded, e.g. the service can only support X number of connections.

If the server-processes that give the service can be reconfigured to manage the requirements from other modules this should be described. This information should include which parameters that has to be changed and where to change them. If the processes can be distributed in order to increase performance this should also be described.

A procedure for determining how much capacity there is left in a service should be described. If the description is lengthy, a reference to a document describing the procedure should be given.

17.5.3 External communication

External communication is often very complex and can cause conflicts when different types of standards are used in the same system. A description of how the communication is defined should therefore be included in this section.

A procedure to determine how much capacity that is left in communication channels be given, either by direct description or by indicating to a document describing the procedure.

18 Installation

In order to make the installation procedure more efficient the installation should be made on modular level instead of as today on CXC level. This chapter is not an attempt to describe the installation procedure completely. It should instead be seen as a starting point for discussions.

To make the installation as efficient as possible the number of errors made during the installation has to be as small as possible. It is thus important to make the installation as automatic as possible and there must be as little operator interaction as possible.

All parameters should be defined in a file or a database that is common to all installed modules. All parameters should also be defined only once and they should have unique names. That way the errors that result from misspelling a parameter, e.g. severname in one file but not in another will be eliminated. There will also be no need to edit numerous files. The fact that all parameters are gathered in one file has the advantage that there is only one place to search for possible errors. It is also easy to see how the whole system is configured just by looking in the configuration file.

The actual implementation of how to handle the parameters may vary. Either a plain file could be used or Sybase could be utilised. Sybase is powerful but using it might be a little bit of an overkill. The central parameter database should be used as much as possible. The configuration parameters could be put there, entries in login files and so on. The parameter database should be equipped with a utility that detects conflicts between the parameters of the various modules.

With all the parameters defined centrally in a database it is easier to design the installation procedure so that all parameters are entered beforehand and then the installation is completely automatic, i.e. no parameters should be entered during the installation phase. This makes installation easier since all parameters can be set at home and the file can then be brought to the site.

It should not be possible to change parameters that are predefined. In fact those parameters should not even be seen when installing. This will keep the operator from inadvertedly changing something that should not be changed.

There should not be any workarounds in the installation procedure of a module. Workarounds reduce efficiency, are time consuming and cause errors.

All modules should be given a default configuration that can be used in most cases. That way most parameters can be left as is only a few need to be changed. Installation becomes more efficient and less errors are introduced.

It is important that one person can install a module without detailed knowledge of the whole system. That way the whole system can be larger since a few people can be experts on different parts of the system.

It is important that the installation of modules is defined in comprehensive design rules, to allow error reports to be sent to design, so that they can correct their errors.

There are some drawbacks to using an automatic installation, for instance if an error occurs the whole installation prior to the error must be re-run, or if the error is only reported in a error log the error can cause even more errors later on. This could be handled by a more sophisticated installation script, but this is costly and is highly dependent on the software that is to be installed.

19 How will modules affect documentation?

Here is a survey of the documents that SD may come in contact with. If the document is affected by the new modular handling of TMOS a short explanation of how and why the document is affected must be included. Most documents at Application Line level will drop down one level to module level. The CXC level documents will be less important for the SD department as Design will have to take more responsibility for creating good documentation at higher levels. In order to be able to reconfigure the system in various ways it is necessary to write a large number of reconfiguration documents that describe how to change the parameters that resize the resource. In some cases a more complex description is needed, e.g. when adding a new server. If the description of parameter changes can be kept short, it should be included in the module description document and not in a separate document.

19.1 Appl. line composition

109 21-n/AOM Product Rev Info

Will be written both for modules and parameter lists. There will be a need to document revisions of both modules and parameter lists.

9/1095-n/AOM “Correction Survey”

2243-n/AOM Question List

131 62-n/AOM Ordering Information

see section 19.4

1531-n/AOM Installation Instr.

This document will only be written at module level.

1532-n/AOM Test Instruction (Installation)

This document will only be written at module level.

1057-n/AOM Engineering Data

This document will only be written at module level. In the report this document is called ‘Module Description Document’ and is an extension of the previous Engineering Data Document. The difference is that dependencies and resources are also described.

n/190 59 n/AOM Parameter List

The parameter list will actually get a more important function in the world of modules. It will be written for standard configuration systems and there serve as a new AL concept where a combination of modules have a standard configuration. It will also be written for each site, but then just as an update of the system parameter list. In some cases a special configuration of a system is needed and a unique site specific parameter list is engineered and written. The parameter list should be divided into two parts, one with parameters that often has to be changed and one with parameters that rarely has to be changed. Every module has its own parameter list that is written by design to fit most applications. It is from these that the major part of system parameter list is produced, the rest is engineered by the SD department (parameters that are dependent on dependencies).

19.2 Appl. unit composition

131 62-FAB Ordering Information

see section 19.4

1/131 62-FAB Ordering Data

see section 19.4

109 41-LZV Document List

19.3 Exec. unit composition

1510-CXC, CXA Manufacturing Instr

190 06-CXC, CXA Generation Info. “Makefile”

109 89-CXC, CXA Container File

109 21-CXC, CXA Product Rev Info

1531-CXC, CXA Installation Instruction

This document does not serve any purpose for the SD department since installation instructions for modules will be engineered by Design.

155 18-CXC Application Info

This document is not affected by contents but by usage since this document will not be used by the SD department when creating Engineering Data documents. It will probably still be appropriate when doing special configurations of systems and when tracing errors in the installation.

1545-CXC Fault Tracing Info

This document will be used when looking for faults in a CXC. A fault tracing document on the module level is also useful.

19.4 Ordering information

The documents that describe which parts of a system can be ordered separately and which parts must be ordered together will only be needed on one level, the module level.

The information in the module description document is sufficient to determine what combination of modules and options within them are allowed. Since few dependencies between modules exist there will not be a great need for this information.

There might, however, be a need for a condensed table of all the dependencies between modules.

20 A completely different solution

This report gives one solution to the problem presented initially, namely that the cost of handling TMOS is too high. The solution presented is believed to be the best but other ways of solving the problem do exist. This chapter gives an introduction to how the problem might be solved in another way.

In this report the solution of the problem with handling TMOS is based on avoiding the problem, i.e. that by foreseeing all possible problems that might occur in a system problems can be detected and solved by allocating enough resources for all tasks that must be done. Even if this might be an approach that is fairly simple to implement, this solution has a number of drawbacks.

One problem is estimating the load on different parts of the system. There are simply so many factors that need to be taken into account that precise models are virtually impossible to create. Since the system is so complex, it is also very hard to predict the behaviour of the system. Another factor that makes is hard to predict system behaviour is that TMOS is based on UNIX which is inherently difficult to predict.

There are other solutions that are based on a completely different concept. Instead of predicting and solving all problems beforehand, problems are detected when they arise and resolved then. Since this might be the way to go in the future, this solution will be discussed a little more closely.

All resources keep track of their own load; when the load increases above a certain value the resource automatically expands itself. For example, if a database is becoming full or if the load on one processor in a multiserver/multiprocessor installation becomes too high compared to the rest of processors processes are automatically migrated to processors with spare capacity. If a resource such as a communication link is missing, this is reported to the operator so that the resource can be installed.

When another module is needed in the system that module is simply thrown in and the rest of the system adapts so that the new module will work. If the module needs more resources it will tell the operator so. It will probably still be necessary to have some kind of documentation that describes which modules can be installed separately and which modules need other modules in order to avoid embarrassing situations when one comes to a site to install a module, only to find that the new module depends on another module that one of course did not pack. It is also possible to set warnings so that the system warns that a resource is close

to running out and it may be time to get more of that resource, for example disk or CPU power. The main point is that there will be no need to predict how the system is going to put load on the resources and therefore no need to dimension the resources. And thus there will be no need for documentation describing this.

There will, however be a need for mechanisms for automatically moving resources around in the system. These mechanisms can also be utilized to create high availability solutions. Consider a system with four servers with all resources doubled. If one server fails this will be detected and those resources that resided on that server will be duplicated from their still working copies and spread around in the system. The system will still function but on only three servers. When the faulty server is replaced or if one wants to add another server to a three server configuration, that server is simply hooked up, the system detects it and processes and databases are automatically migrated to the new server. The conclusion is that since there is a need for a mechanism for moving resources around, that mechanism can be used to add resources while the system is running.

The main drawback is that this is hard to implement. To move processes around transparently, support for this is probably needed in the operating system. The database server must also have support to move databases around while the system is running. The database system used today does not support this. There are also many parts of TMOS that need to be redesigned.

There is also some overhead created when monitoring the various resources. This might not be a serious problem since hardware is relatively cheap and all that has to be done is to add extra hardware to give enough power to handle the overhead.

Another problem is that the system becomes even more indeterministic than it is today. It is not easy to handle the system today, but at least the processes and databases stay on the same machine. If they are allowed to move around it will be more difficult to determine how machines are going to perform and conducting tests will also be more difficult.

In conclusion this is a solution that can result in greatly simplified handling but it requires a considerable amount of redesign at high cost.

21 Conclusions

The problem when handling TMOS today is the time the various activities take. One time-consuming task is the generation of a new AL from the FABs. Another task that takes considerable time is the creation of a customer-specific system that can not be derived from an AL. Testing the system is also time-consuming, since there are no methods to determine which parts of the system affect each other.

To solve the problem with handling, modules are created. Modules should be about the size of three to seven FABs. This results in a reasonable compromise between flexibility in a system and the number of modules that have to be handled. The functions in a module should be related to each other, enabling practically all dependencies between modules to be avoided. As modules grow there might be a logical gap between the functions in the module. The module should then be split in two.

All dependencies between modules are described using resources. This makes it possible to describe two types of dependencies. One type occurs when two modules depend on the same resource. The other type occurs when one module is dependent on a resource in another module. If all resources are described, all problems can be detected, but this necessitates a great amount of documentation. By describing only bottlenecks, not describing obvious resources and by adjusting the size of the various resources, the amount of documentation can be reduced while almost all problems can still be detected.

Options are allowed in modules, but the interface of the module is kept constant. This means that even if an option is not selected, resources for it are allocated. Options are distributed and installed together with the module, but to be able to use it a key has to be given. This way pricing will be simplified since handling and price are less related than today. It will also be easier to sell upgrades. The way options are handled imposes certain restrictions on options. An option may not depend on another module and it should be possible to install a module even if an option depends on hardware that is not present.

In order to handle the modules efficiently a module description document is created. This document is described in detail. The document is divided into five sections.

- Short description: A short description of the module.

- Table of dependencies: A short description of what the module is dependent on.

- Dependencies: Describes which resources the module is dependent on.

- Engineering data: Describes parameters that can be used to configure the module.

- Available critical resources: Describes the resources that the module provides.

This module description document then replaces today's engineering data document.

The module description document can also be used to reconfigure resources (e.g. databases or server processes) when the load on them is increased. If the reconfiguration of a resource is very complicated a separate document has to be written to describe the procedures.

To simplify handling even more the platform is preconfigured. The platform may not then need to be changed when a module is added to a system, since all the resources that are required may already be correctly configured.

The result of the engineering process is a parameter list. The engineering work is done when a system that consists of a specific combination of modules is created for the first time. When another system that consists of the same combination of modules is created again there is no need to do all the engineering again since the parameter list can be used. It does not matter, however, if the options in the modules are present or not, which gives the parameter list some flexibility. This parameter list replaces the AL of today. Most systems can then be derived from these parameter lists. It is only in special cases that the engineering will need to be done again.

Modules are preconfigured by the Design departments. Installation can the be simplified in most cases because it is not necessary to change parameters. The installation of a module should be complete, i.e. no additional configuration of the whole system has to be done once the module is installed.

Modular handling puts restrictions and demands on modules, options, the platform, the installation procedure and documentation. Some restrictions must be met in order to make the model function correctly, while others result in simpler handling if they are met.

The advantages modules are that engineering, handling, installation and testing is simplified. Engineering will be simplified because there is no need to search through many documents to find the relevant information and there is no need to go through the whole system to add a module. Because the system consists of a few modules the number of documents will be kept low. Handling will be simplified because there is no need to generate a complete AL. Installation is simplified because all parameters are defined beforehand and only once. Testing is simpli-

fied because it is possible to predict what parts of a system need to be tested when a new module is added. It is also easier to create new revisions to add functionality or correct errors because it is not the whole AL that has to be changed, but only one module.

The drawback with modular handling are that since more flexibility is created certain tasks, such as configuration of a new system, may be more complicated than they are today. These problems can, however, be handled with standard configurations of a system. Another problem that may arise is that the document describing the module may be hard to create because it is written by people not have an overview of the system and because formulas for dimensioning resources may be hard to create.

The complexity of the TMOS system and its many configuration possibilities makes it hard to the system to modular handling. A great effort has to be made to simplify the structure of TMOS, perhaps by reducing the number of parameters that can be set and making FABs and modules even more atomic.

Diary from the thesis project

This is a description of the different activities that we have done during the project. It is in roughly chronological order.

1.0 The introduction

After some introductory phone calls we were finally able to begin the project. We started out with some introductory meetings with different people that had interests in the project. Everyone gave their opinion about what had to be done and everybody we talked to said that ‘this is really interesting, but it is difficult’. And everyone gave us the impression that there were really big problems that had to be solved.

The first thing we had to do when we first encountered TMOS was to learn more about the system. In order to do this we read all kinds of documents, mostly the ones describing the whole system. Reading documents would then follow us during the whole project. There are not many documents describing the whole system, but since those documents averaged around one hundred pages each we had more than enough to do.

Since fiddling with documents were going to be one of our main tasks we also had our firs contact with PRIM. PRIM is the system for handling documents in the Ericsson world. It is probably a very powerful tool, but it is not userfriendly. Our first impression was not improved by the fact that PRIM is only a tool for finding documents and that another tool is required to retrieve the documents.

In order to get some hands on experience we spent one afternoon at the support group playing around with one of their TMOS systems. It was an XM system, and those does not have that many different menus, nice little icons and buttons to push. In fact a normal word processor can do many more things, at least that was what it felt like. We went home with a feeling of that this was going to be easy. We also felt a confused, what was the problem really?

2.0 Digging into the problem

Another task that is part of the standard introduction to TMOS is installing a system. We were assigned a couple of old computers that nobody used. These two old (at least a year) computers were ours to use for our project and we begun the installation. In the beginning we had trouble with the console terminal and the little nifty editor vi. We also had problems with our hard disks and though we reformated them they would not work. Not until a service technichan from Sun came we were able to solve the problem.

When we were halfway through the installation somebody came by and asked why we were using that old document revision. It turned out that the revision of the installation description that we were using was rather old and we changed the revision of the installation description. In fact one of the big problems that we had during the whole installation was to get the right revisions of documents, software and hardware. Our troubles with revisions might have caused some of the errors we encountered later.

We were able to complete the installation thanks to all the helpful people around us. But when we had done it turned out that not much of the system were up and running. A rough estimate is that about half of the system worked. In the beginning we blamed the external communication. Since our computers were not connected to the outside world we thought that if some parts of the system could not communicate that would create some kind of snowball effect that would cause other processes to fail. Thus it would be perfectly normal to have only about half the system start. We spent one happy day in that dream. Then somebody told us that we should be able to get the whole system up even if we did not have any connection with the outside world. Then we spent two days trying to locate the bugs. But after two days we gave up and found another worthwhile task to do.

Since we thought that we were going to study the dependencies within TMOS that was where we started. We did this by studying the individual installation documents for each CXC. A CXC is a small building block that consists of one or many processes. Anyway in the installation document there is a chapter where information on which CXCs that has to be installed and running in order for this CXC to be installed and run correctly. We managed to find a list of all the CXCs that is included in a XM system and by using the installation manuals at the department in combination with PRIM we managed to get information about how all the different CXCs were dependent upon each other. We decided to draw a map of those dependencies and ended up with a huge map that sure looked complex. We were happy since this really seemed like a complex matter. We had found our problem.

But when we grouped the different CXCs into FABs and grouped all the CXCs in the platform together there were not many dependencies left. When we discarded all the CXCs that belonged to CMAS there were only four different dependencies left. But that seemed like a good result, there were not as many dependencies as expected so grouping of the different FABs could be done without too much effort. We left a note on our instructors white-board telling him that our job was done and left for the weekend. We didn’t know at the time that we were on the wrong track. We wouldn’t find that out until weeks later.

The next week we talked to our instructor and discussed what we were going to do next. Together we decided that we were going to study the dependencies between the CXCs better. What did they consist of really, what was the cause of the dependency. We went hunting for information through the building. We found interworking descriptions, but the information they gave was too detailed and not really suited for our needs. Other documents we studied gave too much and too detailed information while other gave little or none information that we could use. We also talked to a couple of people and that led us to Mr. Jonas Udén. He had done something similar to what we had done about two years ago and he seemed to be the ideal person to talk to.

We met Mr. Udén one morning and he told us a few thing that we already knew, namely that there is no written information anywhere that describes the dependen-

cies among the various CXCs. Mr. Udén had mapped the processes and the dependencies among them. When he had done research he had had to talk to the various programmers and talk to them about how the processes were connected. When we left he gave us a copy of his documents and we thought that we had found a gem. We went back and started putting together all his maps into one big map describing the complete system. When we grouped the CXCs together into FABs we found almost, but only almost the same dependencies that we had found when using installation documents. There were many different factors that could account for these differences. The map was old and the dependencies described in the installation documents might be of another type or the person who wrote the document might have included a couple of extra CXCs, better safe than sorry. It is also possible that we made a few mistakes along the way.

We knew we were on the right track and so we decided to try to put together a template for describing the various dependencies in the system. What level were we going to put the description on. After long discussions we decided to put the description on CXC level. That was done because we wanted to be able to trace the dependencies back to CXCs and with the process map one should be able to trace the individual dependencies back to individual processes. We put together a template and distributed it to a few persons to get some feedback. To get the answers back we had to wait for a while. The people at Ericsson are busy.

3.0 Adding other tasks to the project

It was during this period of temporary hold in the project that we found out that we needed something else to do so we were given another assignment that we could fiddle around with in our spare time. When the handling of a TMOS system becomes easier it will be easier to expand a system. Then methods will be needed and our task was to develop such methods and tools if there was a need for them. We were even allowed to write our own specification. Our instructor was really busy at the time and spent most of his time meeting with other people, so he did not have time to think about his diploma workers.

In order to be able to develop methods for expanding TMOS systems we needed a working system to experiment on so we went back to our system and tried to get it to work. It was really a basic system and it was supposed to be easy to handle, at least compared to the other TMOS systems that were around. But we were out of luck. When we had spent two days trying to find the error we gave up and decided to try again from the beginning. Fortunately we had a tape that contained the platform, so we would not have to go back to the very beginning. We made the installation and after trying to shake the bugs out of that installation we concluded that it was probably some errors in our first installation of the platform too.

After some discussion we did the whole installation again from the beginning and managed to get the system up and running. Of course we had some problems this time too. One of the things that played us a little trick was the installation of the authority database. After waiting for a couple of hours we decided that something

must have gone wrong, so we stopped and asked what we had done wrong. It turned out that nothing was wrong. Installing the authority database takes about eight hours on our slow machines. But finally were ready take care of the migration of databases.

Reactions on our suggestion about interface description started to come in and we soon saw that we had chosen the wrong level. A description on FAB level was obviously what people wanted. But we got a lot of constructive feedback so we were able to do some serious revising of our document template. We also saw the need for documentation on FAB level. Partly because there are not many descriptions on FAB level and with the size of the document that we had proposed it would be unrealistic to write on for each FAB. We were happy, we had finally found our problem, we were making progress and satisfied with life we gave the document to our instructor and went home for the weekend.

We had a meeting with our instructor the next Monday. He was not entirely satisfied with what we had done. He did not want a description of the dependencies in a TMOS system. No ! He wanted something that would ease handling for their department. We were lost, we had lost our problem. Or rather we had solved the wrong one. We had been on the wrong track for more than a month and now the track ended.

4.0 Starting again with a different approach

We went back to our office. Yes one of our colleagues had left for vacation and he lent us his office. We sat and discussed back and forth and forth and back. After a couple of days we could se that there might be a point in that there is a need for larger handling unit than FAB. We ran into a lot of problems but finally we found a solution that might be viable. Then we rewrote our document and added sections about modules and the handling of those.

We gave our instructor the new revised copy and asked him to read it and then we dug into our second task of migrating databases. One thing that had to be done was to add a disk to an existing system and then migrate the databases onto that disk. It wasn’t really a problem, we had the disk and everything we had to do was to hook it up to our system and then we would be on our way. Anyhow anybody that has worked with computers knows that nothing that seems simple, really is simple. We needed expert assistance, or at least someone who could point us in the right direction. So we went roaming through the house. We were lucky since if was Friday afternoon at about 4 pm and the guy we needed to talk to would leave for vacation the next week. He was not as happy as we were, but he gave us a hand and finally we got our disk to work.

We spent time on going through the various ways to move a database server and made progress. Although we made a mistake now and then and then we had to restore our server from backup tapes. That is easy to do, but the problem is that it takes a couple of hours to do and mistakes are often made early in the day. So we

spent some hours drinking coffee and discussing the inns and outs of the world in general. But despite these problems we were able to finish the four different methods and write an instruction for how to migrate databases onto another disk. The next task would be to add one server to a oneserver system. We started to install a second machine and install a databasesever on that machine.

But things happened that we had no control over. Olav caught a cold Tuesday before midsummer and didn't return until the next week. While he spent his time in bed with tea and honey Henrik continued with the second databaseserver and he also had a meeting with our instructor. Our instructor told him that we were on the right track but we still needed to connect what we were doing with the rest of the Ericsson world. We should also describe the dependencies more closely. Henrik left to celebrate midsummer in Gotland. While he was there he fell off a rauk and sprained his ankle.

Limping and coughing they returned to work on monday morning and we started digging in to the problem on how to describe dependencies. We spent Monday and Tuesday thinking. Thinking meant sitting in our room and discussing and drawing on the white-boards. At times there would be hectic writing on the board and loud discussions and at time we would just sit there and think. Unfortunately it was only on those occasions that people would walk by and look in. They must have thought that these two guys does nothing productive, what are they doing here. Anyhow the rest of the week we spent writing more documents and revise the earlier documentation. Our bump test of the our documents started to give good results too. A bump test is simple to conduct. One just drops the document on a table and if it makes a pang it is good. The louder the pang the better the documentation.

One of our problems was how we were goung to fit our work into the rest of the TMOS system, were we going to create a new ABC class or could we use an existing one. Or would it be so terrible that one ABC class had to discarded. To understand the problem one has to know about ABC classes. ABC classes is what keeps the Ericsson bureaucracy going. It is a way to classify everything that Ericsson does. Two things are produced: documents and products. Both of them has their own way to number things, and those two things are related in some magical way. The numbers are mystical themselves with lots of slashes numbers and letters. There is a theory that this way of numbering things is created just to keep company secrets and guarantee that only the best may be employed. Those who cannot understand the numbering system are not fit to working at Ericsson.There is a simple spell to find out how things are related: PRIM. Unfortunately the results of this spell is a little bit unpredictable. Anyhow ABC classes has kept Ericsson going for more than half a century and they will probably remain until the final day. If we were going to create a new ABC class that would cause disturbances around the world and the decision would probably take years to make. So without even considering other alternatives we decided to let our document replace the engineering data document.

The rest of the week was spent writing down the results of our discussions. During the writing process we discussed those issues that obviously were not clear enough to put in writing. Writing down ones thought is a good way to really find out what one is thinking. Especially if ones thoughts are unclear. We wrote different sections each and then checked and edited each others sections. One thing that we found out was that there were other entirely different solutions to the problem. Not the problem that we were told to solve, but the problem that Ericsson had. The problem really was how to bring the cost of handling TMOS down, but we were told to find a way of dividing the system into modules. But we found out that there were other ways of keeping the cost of handling TMOS down. Unfortunately that would require redesign of the whole system. Anyhow by Friday afternoon we thought that our document was fairly complete and the bump test gave good results too, so we gave our instructor the document and wished him a happy weekend.

5.0 Job for experts done by novices?

Monday morning we went back to our project with expanding an already existing system. The task of adding another server seemed fairly simple. Just connect the server physically, install UNIX and the database server, move the databases and change a few parameters in the system. The next morning we were completely out of ideas and our whole thesis work came to a temporary halt. Installing UNIX and the database server was straightforward and did not cause any serious problems. Migrating the database was also simple, at least we knew how to do it. Then changing parameters in the system was the big problem. After searching through the whole filesystem for the string “SEOMC1”. We changed all occurrences to SEOMC2 and hoped that it would work. It wouldn’t and after looking through the whole filesystem a couple of times we decided that it is not only the SEOMC1 that has to be changed but also the hostname probably had to be changed in a few places. After a couple of fruitless tries we gave up and turned our attention towards more interesting (?) tasks. Obviously moving databases is a job for those who has worked with TMOS a long time and who knows all the inner and outer working of every little tiny file.

6.0 Are we getting close to the end?

That week we had two meetings with our instructor. On Monday he hadn’t had time to read the whole thing so he didn’t say much, and that fooled us. We thought that this was it. We were almost done. But on Thursday he had had time to read our report and our dreams of a happy vacation until the end of August just sailed away. He told us that we were heading in the right direction, but we still had to do some more thinking. He believed in our theories, but they were still too abstract. What were Ericsson going to use them for, how could they be used and which effects would that have. We tried to point out that in chapter so and so there was a short description on how to use the document, but that was not enough obviously. Our instructor was going to leave for vacation the next week so we had to come up with something to do the next couple of weeks. Our instructor told us that it would be a good idea to have the report completed the eight of august, because then there

would be some kind of meeting in project seagull and they really would need our ideas. At least that is what he told us.

We spent the rest of the week fixing our report and late Friday afternoon we gave the report to Kjell Andersson from the PY department so that he could give us his opinion too. An then we left for vacation too. We only had one week and one day, but we really needed that rest.

7.0 Working but tired

We got back on tuesday. Vacation had been good, but we didn't have too much sleep so we were a little bit tired when we returned. Our first task was to find Kjell Andersson and discuss our report with him. He had read our report. Unfortunately the introduction reveals that we are novices in the TMOS world, so Kjell probably read the beginning and then browsed through the rest thinking something about ignorant students. He didn't say much about what he thought, either he had no opinion or he kept quiet. But he pointed out that we had to place the whole thing into it's context. We had heard that before, and he gave us the name of another guy that had worked on the problem before. We left after a while and got back to our office.

We continued fixing our report up by adding an introduction and a conclusion. We also tried to expand the chapter on how to write the module description document.

We talked to the next guy in the chain, Rune Tedin. He was a friendly and spoke-some guy that talked a lot about structuring of TMOS and different approaches that has be thought of. He gave us a couple of good ideas and what was more important was that he thought that our ideas was good and that they might be usable. He also told us that there had been work done concerning the same thing in the AXE world. And like everybody else he gave us another person to contact.

It started to dawn at us that this project had no end. It is possible to dig in and dig deeper and deeper and deeper for many years, and there will still be another approach to consider and always another person to talk to who has done something similar. The never ending project...

We were not very effective that week. But we spent the rest of the week doing what we always had done. Thinking and writing. And we started to feel the panic, things had to be done in time, and we started to get really fed up with the whole thing. But fortunately it was time for another week of rest and recuperation. The weather was probably going to be fine so we left Ericsson happy as clams.

8.0 End of the report and the beginning of the presentation

When Henrik returned from vacation on Monday the eight of August he continued writing the general parts of TMOS, intended for Chalmers and those who do not

understand TMOS. (Who does?) Thursday Olav returned and we had a meeting with Niklas and talked about our report. Finally it seemed like we were heading in the right general direction. Niklas only gave us a few more things that we had to update. So we spent the remains of that week updating the report. What was really important to find out was how big a module should be. There are many factors influencing the size of a module. Unfortunately we didn't know exactly how much each factor contributed to the cost of handling a module. So we guessed in a more or less orderly fashion. The result was a neat diagram that looked really scientific. We could probably have developed advanced formulas as well, but since this is a serious place we refrained from doing that.

Henrik left for another sailing trip. This time with the big ship Havila. Olav stayed behind and did more work. One chapter had to be completely rewritten, the one about modules and how those were to be created. There were a couple of other things that still had to be corrected. He also spent some time reading the entire report once again. There were a few more errors and a few more things that required clarification. Will that report ever get done?

Olav also started to do all the administration that is required in order to get the diploma thesis accepted. It is not easy, and since the system at school is completely new no one knows anything. But at least he managed to find somewhere to be and a good time for the presentation.

Olav talked to Niklas about that, and Niklas said. - "Interesting" and "when can you do the presentation here. I'd like you two to do it as soon as possible. Yesterday would be absolutely best." So Olav sat down to start preparing for the presentation. Writing slides is fairly quick so by the end of Friday he had a comparatively finished draft. Poor Henrik had no idea of what was waiting for him when would get back.

Chalmers wanted a report and we didn't want to give them one of the Ericsson reports. Although Ericsson have comparatively efficient reports they are ugly. So the best part of Friday was spent battling with FrameMaker to make the document template look as we waned it to look. Unfortunately Framemaker did not have the same opinion as we did, but we won.

9.0 The end

Henrik got back on Monday and we did the finishing touches on the report. And that included finishing the diary. And because the report had to be finished the diary had to end too.

What happened then is that everybody was happy and lived long ever after. TMOS conquered the world and Henrik and Olav became highly paid employees of Ericsson. And as far as I know they are still writing TMOS documentation...