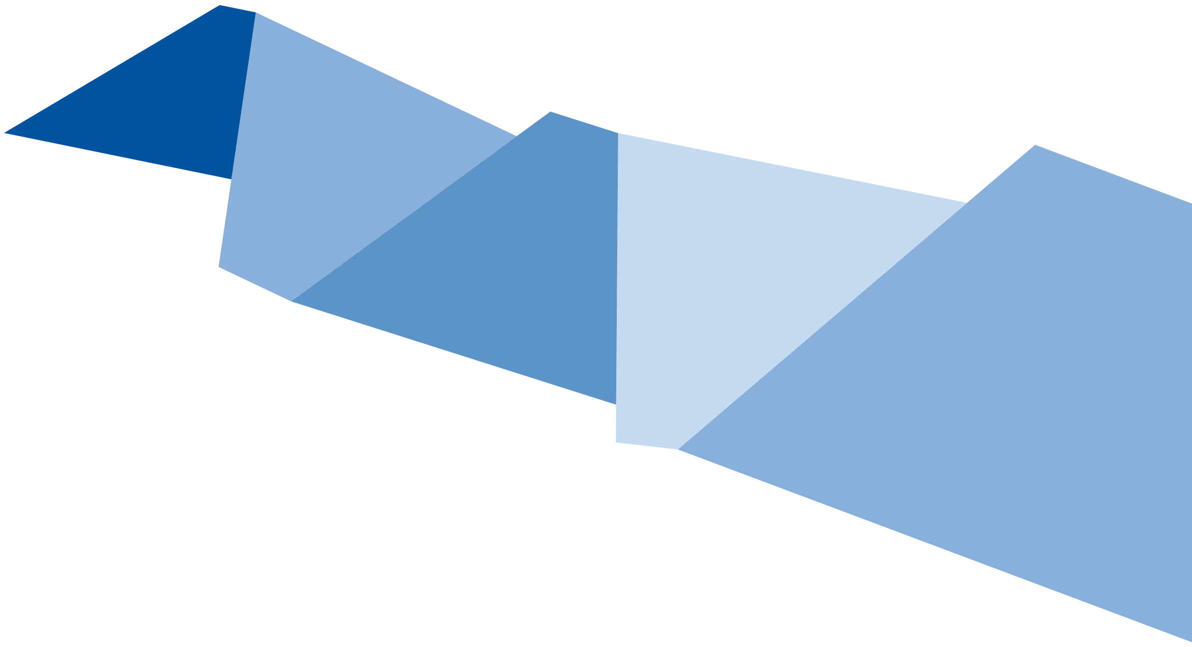

Sequences of Software Architectures

Manfred Nagl

Department of Computer Science

Technical Report

Aachener Informatik-Berichte (AIB) | ISSN 0935-3232 | AIB-2021-03
RWTH Aachen University | Department of Computer Science | February 2021

Sequences of Software Architectures

Manfred Nagl
Software Engineering
RWTH Aachen University, 52074 Aachen, Germany

Abstract

The architecture of a software system consists of different artifacts with hierarchy and other dependency relations between them. This architecture is developed in stages. It starts with an abstract and conceptual form and ends with a concrete form with many details, describing many technical details of the shipped system.

This paper deals with the specific question, how these stages of the architecture have to be organized. What is the order of these architectures and what are the relations between them? The order is connected to the methodology of the design process and to the properties of the resulting design, as e.g. adaptability. The underlying architectural language should be able to express these relations. Two examples, one from business administration and one from embedded systems, are discussed.

Keywords: software architecture modelling, different stages of the architecture and mutual dependency, order of the stages and adaptability, desirable uniform architecture language and methodology for the stages

1 Introduction

What is a *software architecture*? It is a collection of artifacts, which (or some of which) correspond to different goals, as overview/ detailed, according to hierarchical relations as e.g. layers, according to different degrees of detail, according to the art of presentation (graphical or textual), according to the type of description as static (structure) or dynamic (execution), etc.

All these artifacts are related to each other and are aggregated in the *architectural configuration*. This architectural configuration is part of the *overall configuration* of the complete development process, which, in addition, contains all the artifacts belonging to requirements, programming, quality assurance, documentation, and management.

This is rather similar in all *engineering design and development processes*. What we call architecture in the software business is there called conceptual design or modeling, what we call programming or implementation is there called detail design or engineering. This view of the overall configuration, having different *technical parts* and *others for management, documentation, or quality assurance*, exists also in the field of architecture for buildings and civil engineering. There, the overall configuration is called "building information model" (in short BIM /ET 08, KN 07, Kr 07/).

This paper concentrates on the architectural configuration and there on the statement that there is not only one architecture. The architecture runs through different stages from abstract to concrete. The question "What are these *stages* and *how should they be ordered* in the design process?" is the topic of this paper. We assume that the underlying architecture language is able to express a variety of concepts.

The architectures of the series belong to different stages of development. These different stages correspond to different *aspects and views*. The *order*, in which these architectures appear, should be *carefully organized*. The architectural *language,* in which these different architectures are denoted, should be able to *cover* these *different aspects*, it should be universal /Na 21a/ in the sense of integrating these different aspects and views. This avoids that different architectural languages are used, where it is not clear, how they are combined w.r.t. syntax, semantics, pragmatics, and methodology.

This paper is as follows: We start in section 2 with examples in order to show the different stages for two scenarios, which belong to different application areas. Section 3 discusses the rules to apply on the way to find the suitable order of architectures. Using these rules, sections 4 and 5 explain the order we have found for the scenarios of section 2 and sketches the contents of the different architecture stages. Section 6 sums up, which different aspects the universal architecture language should cover, thereby defining the requirements for the architecture language. Section 7 summarizes the findings of this paper.

2 Scenario Examples

Before going into details, let us discuss *two example scenarios* as introduction to the main part of this paper. Both are valuable scenarios, many others are also possible.

A business administration application

We regard a *business administration application*, e.g. a system in an *insurance company*, which supports employees in preparing an insurance contract for a client, who is interested in getting such a contract. We do not look deeply into the application system, we only sketch the different stages, the architecture of the system could go through, see fig. 1.

We first assume that the *system is new*. The architecture is carefully designed, but in a rather abstract way. So, the services are planned, the main components have been found, and the system is divided into different layers. We call the result the *logical architecture* as it is rather abstract, thus reflecting an idealized built plan. This is the first step.

In the next step it might be useful to precisely specify at least some components, for example the *components* which are mostly *importan*t for the success of the system. Thus, we define their *semantics*. It might be desirable to specify the semantics of the whole system. However, this can get complicated or impossible (as formulating the taste of the UI interface or non-functional parameters of the requirements make it difficult). Therefore, only the semantics of some of the main components are defined formally. This is done by annotations to these components. This is the second step.

Such a system runs on different machines. Client machines process the inputs of the company employees, i.e. to check them for plausibility or mutual dependency. The main functionality is executed on a mainframe, possibly on different mainframes by reasons of flexibility, redundancy, safety, and security. The underlying data are stored on one data server or different ones, again by reasons of redundancy and security, being located behind thick concrete walls. *Distribution* is described by *annotations*. A methodology can help later by *realizing the distribution* of the system (deployment), using a specific platform. We put these two tasks together and regard the distributed and deployed architecture as step 3.

Distribution induces *concurrency*. For example, different parts of the system might be executed concurrently. The accesses to common resources, therefore, have to be synchronized. Although the whole system is not concurrent by nature, concurrency here comes with distribution as a technical aspect. The result is step 4.

Finally, it might become clear that the system must be modified in order to fulfill the demanded efficiency parameters. So, *efficiency transformations* are used. They usually are placed only, where absolutely necessary. We get step 5.

Summing up: We started with an abstract system's architecture, and with an understanding of modular architectures as usual /Bö 94, GS 94, Le 88, Na 99, Na 21a/. After 5 steps - logical architecture, adding semantics, determining distribution and deployment, adding induced concurrency, and making efficiency transformations - our example system looks very differently. We call the result of all these transformations the *physical* or concrete *architecture*. This architecture describes, how the final system looks like. The regarded example could even have run through further transformation steps.

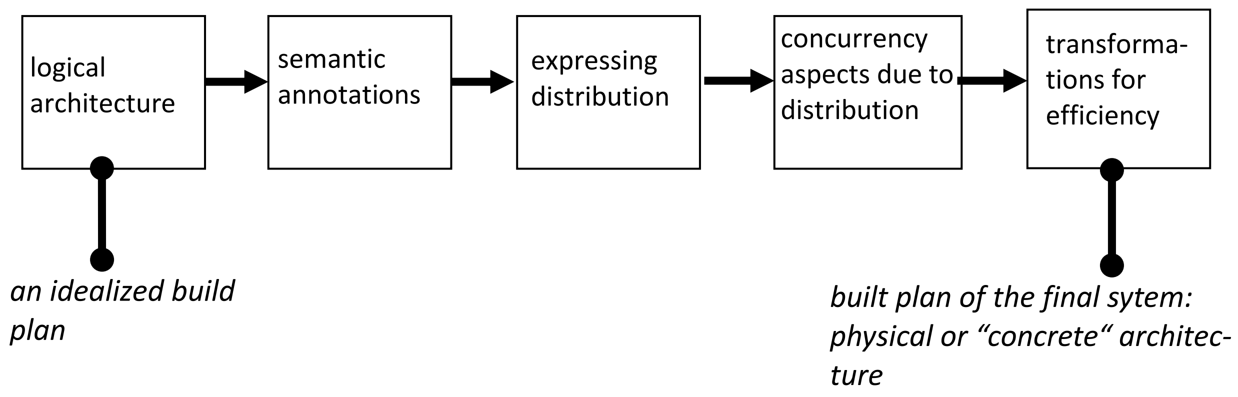
logical architecture
semantic annotations
expressing distribution
concurrency aspects due to distribution
transforma-tions for efficiency
an idealized build plan
built plan of the final sytem: physical or "concrete" architec-ture

Fig. 1: Architecture sequence for an example from the domain business administration

Now we change our scenario. We start with an *existing* and *old system*. This old system was developed without an architectural design. Therefore, the first step is to find out what the architecture of the program system could be (*reverse engineering* /PD 07/). This result is usually not a clean architecture.

The then following step finds a new and suitable form of the architecture (*reengineering*) and tries to modify the program according to this new architecture. The architectural and program modifications do not try to make the system completely new. We are glad, if we have eliminated the most important weak points and if we have a chance to make future modifications in direction of extending the system, integrate it with other systems, and alike.

We call this approach *modification in the large*. It was shown that by reverse and reengineering we get rid of many unnecessary technical details /Cr 00/, which hinder adaptability and further maintenance. Now, the above steps of fig. 1 can possibly follow.

Embedded systems

As second example we regard a possible *architecture sequence* from the domain *embedded systems*, see fig. 2. The first step again is to design the logical architecture, which is free from details. Then, the sequence of changes can start.

Embedded systems are usually concurrent from their nature. So, the second step is to specify which parts of the architecture may run concurrently. Thus, we add annotations which parts are *processes* and which other parts need a *synchronization* protocol /BF 95/, as they are accessible by these processes.

Embedded systems usually run for ever. Thus, they have to be explicitly started and they have to be explicitly shut down. Starting means to wake up and start the internal processes and shutting down to finalize them and to stop them. Therefore, *explicit start* and *explicit stop* need further technical processes and further communication in order to do the job. The design of that part is done in step 3.

An embedded system may also need an *emergency handling*, which immediately shuts down the system in dangerous situations. This avoids that the underlying technical system (a chemical plant, a production line, etc.) gets damaged. The emergency handling cares about these situations (pressure too high, temperature out of limit), shuts down the usual execution in order to care about these possible damages. Therefore, it acts according to the scheme to minimize the damage. The result of step 4 includes this emergency handling.

Embedded systems usually run on a distributed infrastructure. This *distribution* has to be specified. So, we define distribution lines and the *deployment* of the different parts in step 5.

As above, due to distribution *technical concurrency* may come up (further processes, further synchronization), which is formulated in step 6.

Finally, having the system deployed and integrated and carried out the first efficiency test, it becomes clear that some efficiency transformations must be done. If they are not restricted to the bodies of components, then the architecture of the system has to be modified in step 7.

At the end of these transformation sequence, we see again how the final architecture looks like. This final architecture shows all technical details of the system which is shipped to the customers.

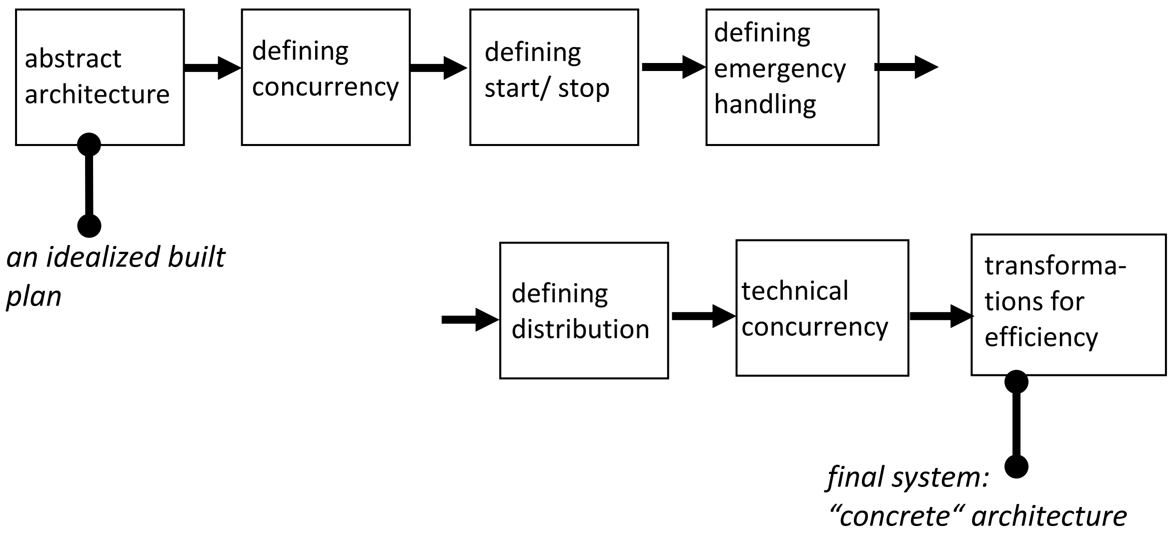

Fig. 2: Architecture sequence for an embedded system

Using processes and synchronization has advantages: We can better control the concurrency at runtime. The disadvantage is that this type of development is rather complicated. Therefore, in practice solutions are built, which often use a *fixed scheduling* by *fixed control loops*. In this case, a control process masters all the interactions of the child processes. This usually makes handling of start/ stop and also emergency handling easier. By using fixed control loops, we get a different architecture in step 2 and also a different architectural sequence. We do not discuss this here.

Summing up: Again, similar to the BA example of above, we get a sequence of steps, starting from the logical architecture and ending with a physical architecture. Comparing to the steps of fig. 1 we see: The steps are different and also how they are ordered.

In the above discussions of fig. 1 and fig. 2 we have a *linear* development on architecture level from left to right. The architectures are developed in a *top-down* manner, from abstract to detailed. This is never the case in practice. There, we find backtracking steps (going back to try something different) and iterations (cycles until the right solution is found). Therefore, the above discussion describes an ideal situation.

Again, there might be *further steps*, before we can start with the abstract software architecture of the embedded system. If the automation and control software is for a new and specialized chemical plant, the *design of this plant* has to be started in advance. It also consists of different steps, from abstract to detailed and concrete.

Alternatively, an embedded system for a specific production line has been built several times. However, it has to be extended. Furthermore, there are demands for quality improvements. So in this case, some *reverse and reengineering steps* have to take place before the *extension*. After the extension, the architecture sequence according to fig. 2 can start.

Two further extensions we do not discuss here. (a) One is that we do not develop one system, but a family of systems /Ja 00, PB 05/. Any member of the family will have a different and specific architecture, as that of fig. 2. In addition, there is the *family architecture*, expressing the commonalities of the family. There are specific considerations afterwards to *derive the specific member of the family*.

(b) Embedded systems in the automation and control domain are in close connection to other systems for monitoring, for quality control, for integrating the embedded system into the company organization, etc. Therefore, the embedded system is connected upward to *further layers* building up the automation pyramid /Na 21b/. The architecture of the embedded system is connected to other architectures via *corresponding interfaces*.

3 Characterization: Order, Rules, and Consequences

Technical details should not drive the design

In industrial practice, we find quite often that (*technical) details* drive the development of systems, especially in the case of embedded systems. The problem is that these details might change. In this case, the development and – especially the design – has to be backtracked. Changes occur more often than forward development. So, *changes* of these technical details, cause many *problems* and consequently *costs*. Therefore, the underlying principle of the order in the above sequences is that details do not determine the solution.

This does not mean that details are unimportant. It only means that *details* should be introduced in the *right place* and at the right time. Furthermore, we should mention, that a solution is seldom found in a top-down and linear way, as both figures 2 and 3 suggest. There are backtracking, cycles and spirals, as known in software engineering. Both figures describe the *ideal situation*, which rarely exists in practice.

The *aspects* of the solution, which play an important role in both examples, are structure, semantics, concurrency, explicit control, distribution, deployment, and efficiency.

Order and principles

What are the underlying *principles*, which determine the order of the architectures in the sequences of figures 2 and 3? What is left of what and what is right of what? What are the corresponding *rules* we should apply to determine the order of architectures within the sequences?

(A) From more *abstract to* more *concrete*

The logical architecture is the most abstract form in both examples. Therefore, it appears on the left side of figures 1 and 2. An abstract architecture can have different concurrency solutions, a concurrency solution may have different solutions for explicit start/ stop, and so on. Also, defining concurrency, explicit start/ stop, and emergency handling is left of defining distribution in fig. 2.

(B) From *basic* knowledge *to additions* in form of annotations or supplements

Defining distribution in fig. 1 depends on the logical architecture and there might be different ways for defining distribution for the same logical architecture. To explicitly organize start/ stop, we need the concurrent architecture. Defining explicit emergency handling needs start/ stop definition before, as an emergency case might appear in the initialization. Additions can have quite different purposes: It might be that the addition gives more *preciseness* (e.g. semantics), it might indicate a *way to follow* (annotation), or it might realize a certain way *for a solution* (thereby introducing the corresponding realization details).

(C) From less to more probable to change, so from more *stable to more changeable*

A transformation to improve efficiency is more probable to happen than defining distribution. The latter is more probable than defining concurrency. Most stable is the logical architecture. We see: Stability appears left, changes appear right.

(D) From an *artifact to* a *depending one*

Technical concurrency depends on how the distribution was defined, see fig, 2. Defining concurrency depends on how the logical architecture was designed. Many further examples exist.

(E) From *early to late decisions*

The best examples of figs. 2 and 3 for late are the efficiency transformations. They are placed at the right side, as they should be addressed at the end of the development process: On the one hand, because they should be applied late, as any change induces that they probably have to be done again. On the other hand, all efficiency transformations take all architectural aspects into account which, therefore, have been regarded before.

Consequences

Following the above rules by ordering architecture sequences has some *implications*. We mention two of them.

They *lessen* the *width of backtracking*. Backtracking means to go back to the place where a change has to take place and to modify from this place on. As more probable changes are right of less probable ones we reduce the width of a backtracking step. Looking on fig. 2: If the efficiency transformation was wrong, the probability is high that we take another one. This is more probable than defining the distribution/ deployment differently. This, again, is more probable than modifying the logical architecture.

The order of the architectures *delivers* what is *more general* (more left) or what is *more specific* (more right). Thinking in these categories "general" or "specific" improves the design and development process. The general aspects are concentrated at the level of logical architectures. All aspects of the following architectures at the right should not be covered or predetermined by the logical architecture. This argument can be applied to all the following architectures. For example, defining concurrency should not care about start/ stop or emergency handling.

4 The Refined Business Administration Example

The logical architecture in a sketch

The logical architecture of the BA example was worked out in a cooperation with Aachen Münchener Insurance (now Generali), especially those parts which are described below. The example is an *interactive system*, preparing and completing an insurance contract for the back office of the company

We present the structure of the logical *architecture* here only as a *sketch* to keep it simple enough, see fig. 3. It consists of an UI part, a dialog control part, the main business functions, and the data access to data stored in specific files or in the underlying data base system.
(i) The *UI part* is responsible for putting in the data of the insurance customer. Usually, the data are checked for correctness or plausibility. The other parts of the system should not know anything about the UI layout and other specifics. Only the selected command and its parameters should be visible.
(ii) The *dialog control part* of the system is independent of the UI part, but also of how the business functions are realized.
(iii) The main *business functions* are independent of the way they are invoked (UI) and also independent of how the data of the system are internally stored. They see the data in a realization independent form via a data abstraction interface.
(iv) The *data* are finally stored in specific *files* and/ or in a *data base system*, following a data base approach and stored in a specific way.

The *logical architecture* of fig. 3 was the result of a *reverse* and a *reengineering* step /Cr 00/, which is discussed later in this paper. The aim was to overcome the following weaknesses of the old system: (a) The primitive UI specifics were visible within the business functions, (b) control of the dialog and business functions were not clearly separated, (c) the specifics, how the data are stored were visible within the functions. Thus, the original 'architecture' was by no means an abstract one. The reverse and reengineering steps took place beforehand to get the logical architecture of fig. 1 in the form sketched in fig. 3.

A *semantics definition* could have been used in the example, as shown in fig. 1. The semantics of (some of) the main business functions in a pre and post condition manner could have been specified. Furthermore, a trace semantics for the system could have been used to define the execution paths. Finally, algebraic semantics /Gu 76, LZ 74/ could have been used for some parts of the data interface.

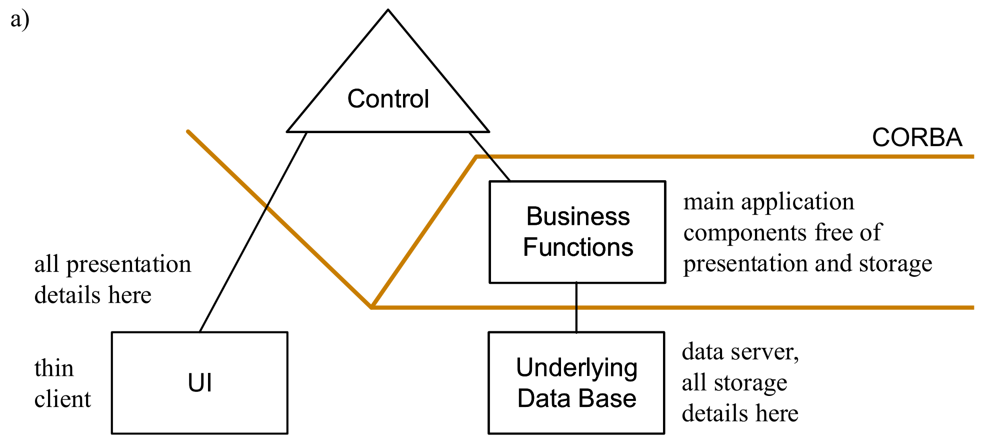
a)
Control
CORBA
Business Functions
main application components free of presentation and storage
all presentation details here
thin client
UI
Underlying Data Base
data server, all storage details here

Fig. 3: A sketch of the business administration system: logical architecture with distribution annotations

In step 3 of fig. 1 the *distribution of the application* are determined. (i) We specify the possible distribution lines between the separate parts by annotations, see again fig. 3 and there the yellow lines. (ii) Then, we add the corresponding components, either basic or generated, according to a distribuition platform, as e.g. CORBA /COR 20/, in fig.3. The resulting distributed architecture is not shown. Finally, we (iii) deploy the different parts on different computers. Typically, the UI part is placed on a thin client, the control part and the business functions on a function server (mainframe) and the data part on data servers (mainframe). This distribution is possible for the abstract architecture as we separated the different aspects of the dialog system (UI, control, business functions, data part). This would not have been possible for the 'architecture' of the original system before the reverse and reengineering steps, due to their unnecessary and tight coupling and corresponding dependency of many details. In addition, it may be necessary that the functions or the data services are replicated and placed on different machines, by reasons of security or efficiency. In this case, we have different function servers and different data servers.

As now different function(s) may be on different machines and try to get access on data services on other machine(s), we get into concurrency problems, although the application is not concurrent from its nature. We called this *technical concurrency* (step 4 in fig. 1). We have to define a synchronization protocol. This, again, is not shown in fig. 3.

Fig. 3 contains a sketch of the logical architecture with distribution annotations. The Corba components according to deployment on different machines are not shown, see /Kl 00, Na 21a/. The same is true for the concurrency aspects due to distribution.

Eventual changes

The *UI part* can have different forms: other UI style, a different UI management system, other commands of similar form, other client (e.g. thick client and extensively checking data). This all should not (or only little) affect the main part of the application. Therefore, the UI part should abstract from these details.

In the same way, the *data part* should be freed from details: How are the underlying data built up in files, which database systems are used, which schema is used for the data, and alike.

There should be a data abstraction interface, which hides all these details. The main part of the program should only know what is necessary from the application side. So, we use data abstraction of different forms.

As already told, we may have different function and data servers, and also the distribution of them on different machines. *Distribution* and *change of distribution* is only possible, if there is loose coupling between the parts on different machines. This loose coupling is achieved by the various forms of abstraction being applied to both sides of a separation line, as sketched above.

Another possibility for a big change is the cooperation of a *program for insurance agents* and the *back office solution* of above. The agent gets the available and interesting information of the back office, enriches this information, and delivers is back to the back office as preliminary and not completely verified. This avoids that the back office has to input these data again. It now concentrates on the verification of the input data and the completion part for the contract.

Adaptability, which was discussed in this subsection, demands for abstraction. Not important details should be hidden, as data, layout, style, UI systems used, schema definition details, etc. This all is *data abstraction* if we define "data" in a more general sense.

Starting with an existing program

As already mentioned, the starting point of our cooperation with the insurance company was not the logical architecture of fig. 3. We now shortly characterize the reverse and reengineering activities. We *started with* a huge and *old Cobol program* on the mainframe, being connected to hundreds of "non-intelligent" terminals, all UI properties and details to be seen in the program, main business functions appear within the interwoven program, and the program knew all the details of the underlying storage of data. How to reorganize this program without to start to program completely new from the scratch. The investigation was carried out within a Doctoral Thesis /Cr 00/

It started with *reverse engineering*, i.e. understanding the big program and finding out the 'architecture' and its essential parts. The aim was to preserve these parts and to reuse them, without using the evident weaknesses of it.

Then came the important part of *reengineering*: (a) building a *new UI part* and make it in a way that all details are in the UI system and the interface of it is completely free of details (how to invoke a command and to get its data) and make the rest of the program UI independent. (b) The *data* part got a *new and abstract interface*. (c) The Cobol program was freed from UI and data access details and separated Business functions from control of the dialog. So, we could identify the main business functions. They got all a new interface.

Reengineering was applied by reorganizing the program and making it better in its internal structure. At the same time, we aimed at getting the code of the old program we can reuse. We called this approach „Reverse and Reengineering in the Large". Big parts of the Cobol program are saved but in another form (without UI details, without data details, clear separation of functions (activities) and manipulation of the underlying data).

All the *reorganizatio*n steps of this subsection *yield the logical architecture* of fig. 3. However, the internal structure of the business functions and the data access was to a big part saved from the Cobol program, we started with. So, they were not completely reengineered.

5 A Refined Embedded Example

Again, we take an *example*, but now from the system class embedded systems. We look on the architecture of this example and give a sketch of some of the *architecture's forms* of the sequence in fig. 2. The example is from the *automotive* domain and contains some simplifications.

This domain is quite specific which makes it interesting for the special argumentation of this paper. In /Na 21b/ the case of embedded software for mechanical or chemical production is studied. In /NW99, NM08/, tools for production processes are studied, which are more standard and not regarded here. We come back to the specific characteristics of automotive embedded systems after we have discussed the corresponding sequence of software architectures.

Software architecture

The *logical software architecture* of the embedded system (step 1 of fig. 2) contains components and their dependencies. It is *abstract*, i.e. the architecture contains no specifics which are determined later, see again fig. 2. Abstractions correspond to functional abstractions or data abstractions, i.e. their realization is below of a *clear interface*. Later on, we discuss that this logical architecture is the result of a transformation process.

Step 2 according to fig. 2 is to determine the *concurrency properties* of the system. This is specified by saying, which components are processes (run potentially in parallel) and determine their cooperation and competition by synchronizations. For example, if different processes try to get access to some data component expressing some state information, we determine a synchronization protocol for the accesses to that component (e.g. mutual exclusion of the operations).

In embedded systems in the automotive domain a *simpler process and synchronization scheme* is applied, as these processes are assigned to different control units to be executed there, see below. The reaction of a process in a certain time is guaranteed by counting the execution times of dependent components and assuring that the sum of these times is less than the wanted execution time. Furthermore, priority rules are defined to avoid that an urgent or critical process is hindered.

A simpler scheme is also used for *start/ stop* (step 3). Whenever the car is unlocked, the components of the body part of the software are waked up, when it is locked, the body part of the car is switched off. Whenever the engine part is started the corresponding components of motor control, drive control, etc. are waked up, whenever, the motor is switched off, the corresponding components are switched off. This scheme works quite well if the different components belong to different areas (body, motor, driving, etc.), which are separated and work together via clear interfaces.

Emergency handing (step 4) is also realized in a simpler way, either by a mechanical solution, e.g. for steering the car, or braking the car. Then, the car has to be stopped immediately and the driver gets a serious warning. It is more sophisticated, if steering or braking is done by wire.

Distribution and deployment

After the modifications described above, the logical architecture is transformed into a distributed solution (step 5 of fig. 2). After the *distribution* lines are identified (analogous to those of fig. 3), the architecture is deployed on the so-called *physical architecture*. This physical architecture consists of control units, connected by buses. The control units are different, on one hand because they realize quite different functionalities in size and also corresponding related requirements, and on the other, as they are determined by subcontractors of the OEM. Control units belonging to one problem area (as motor control) build up a subnet, which might have a specific form and protocol. These subnets are connected by gateways.

The physical architectures can *vary*, even for the same model of a car. There are different variants according to the number of features of this variant. There may be more or fewer control units or such of possibly different computational power, and more elaborated or poorer subnets. So, there are *software architectures* which reflect a *rich functionality* and, accordingly have a *rich physical architectures*. Similarly, an architecture with a poorer functionality can have a poorer related physical architecture.

In any case, there is a balance between the software architecture and its related physical architecture. They are usually *developed "in parallel"*. The physical architecture has to answer physical, efficiency, and security aspects and restrictions. The physical architecture is built up according to *experiences* gained in the *past.*

Different parts of the software architecture are usually *distributed*. The first step is to determine, which parts are to be placed on different hardware. In this first step, we do not say on which hardware the components are deployed. Instead, we determine by an *annotation*, that a subarchitecture or software component is placed differently, but possibly nearby.

In the next step a subarchitecture or software component is assigned to a certain control unit of a certain bus to be placed there, which is called *deployment*. It may be, due to logical relationship (like for motor control), due to efficiency (this part has a higher internal traffic, or this subarchitecture is too much for one control unit), or due to economic reasons (a control unit has open place and available runtime resources) that this assignment is split: one part on this control unit, the other on another unit. Therefore, this deployment is complicated and error-prone if it is made manually. Please recall that this deployment can be different even for different variants of a model.

Deployment is facilitated by the AUTomotive Open System ARchitecture (*AUTOSAR*) *approach* /AU 20/. Here, several programming tasks (as routing through the network, introducing the program parts for a control unit which listen to the bus to find out, what is essential for this unit or, conversely, to put results of a unit computation on the bus again, are done in a semi-automatic way, i.e. the code is generated according to some specifications by AUTOSAR. That is an enormous progress to the situation before, where these code parts had to be written individually.

This approach provides *flexibility in the deployment subprocess*. This flexibility is necessary to handle the many variants of deployments according to the variants of functionality. It also makes backtracking steps easier, which occur after detected errors in a development process. Finally, it also eases reuse in the software between car model families.

Characteristics of the domain and different starting points

There are *various internal problems* connected to automotive systems: The target system (hardware architecture /Ta 12/) consists (a) of about 50 or more control units. These control units are connected (b) by different networks, for example for body control, motor control, drive control, etc. These networks have different protocols and are connected by gateways. Due to the big number of variants (much or few features) of a model, (c) the hardware architecture can vary as well (number of control units). This is also true for the software (number and size of software components. A general problem, therefore (d) is the distribution (together or distributed) and where to be located (deployment). Thus, (e) it is an enormous problem to keep the development of solutions together, from the capability and efficiency point of view. Even harder (f) is the reuse problem, namely to save and apply the knowledge of solutions from one model change to the successor or from model to another model, as OEMs usually start first with more expensive models and later transfer to cheaper ones. Finally, (g) there is an actual trend to reduce the number of control units (less and bigger ones) in connection to electrification of cars. Another trend is to produce uniform solutions in big automotive companies, which have different brands. In the long run, this might simplify the solutions, in the short run it produces further problems as a complex solution has to be transformed to another structure.

How to solve such a complicated problem? It seems reasonable to start with an abstract description of the solution, which is free of all the specific details of (a) to (g). We call this a *conceptual architecture*. This architecture is a network of functionalities/ services, where the connections correspond to data transfer of parameters, to control transfer (subfunction of, or one function after the other), and also synchronization (all next functions in parallel or in a controlled schedule). So, connections represent dependencies, the network is a dependency graph (a dataflow-oriented approach). This network should be abstract, i.e. independent of how to transform functionalities and connections into software and where to place it on hardware.

Now, the *logical architecture* is developed (step 1 of fig. 2), which *transforms* the conceptual architecture to a software architecture consisting of components and relations. This software architecture has at the beginning an abstract form, and gets more and more detailed and concrete, see above.

What are the *transformation rules* between the conceptual architecture and the (abstract) software architecture? A functionality of the conceptual architecture can be mapped on a software component. It can also be that a function is mapped on more software components together with the relations between these components, so on a software subarchitecture. Finally, different functionalities together with their relations (a subarchitecture of the conceptual architecture) can be mapped on one software component. Thus, the transformation is not necessarily one to one. The reason is that the conceptual architecture contains services, whereas the software architecture delivers the structure of a program system, which realizes these services.

Modelling on the level of services (conceptual architecture) and on the level of software components (logical architecture) should have a *similar granularity*. The conceptual model should not contain functionalities of a fine-grained level. In that case, the model is more a too detailed requirements specification or a realization and not a conceptual model. In this case, the software architecture would contain many software components, which compose the realization of conceptual components.

We could have started with an *existing solution*, as above in the BA example. Then, reverse and reengineering steps would be necessary, to improve the solution and to make it ready for modifications and extensions. This was done for an embedded system from the domain communication systems for cell phones in /Ma 05, Mo 09, Re 10/.

In /Me 12/ there is another starting point by looking on different *feature variants* and developing different conceptual architectures for these variants. The start with a variability model helps in finding out the commonalities and differences of variants. The focus of /Me 12/ was to master the variability problem. A further and *bottom-up approach* was taken in /Ar 10, Ki 05, No 07/ for embedded systems in the field Smart Homes. Different eHome systems were built, on top of simple functions/ features offering novel functionalities.

6 Requirements for the Underlying Achitectural Language(s)

If we look on different software architectures /SAD, Sc 13, SEI 10/, there are *obvious requirements* for the used and underlying architecture language(s). (a) The language(s) should contain elements to express all the different aspects and views, as discussed in sections 2 to 5. (b) It should be clear how these different aspects/ views are related to each other. This is not the case for UML /BR 05/, which is a collection of separate and rather nonrelated languages. The situation is much better for subsets /Ru 11, Ru12/ (c) The best is, if the language elements belong to one and uniform architecture language. If we have different languages, e.g. another language for annotations, it must be clear and obvious, to which part of an artifact the annotations belongs.

Within the architectures discussed above we find different parts /Na 90-20, Na 21a/. The biggest and basic part belongs to *modelling* the *logical architecture* and there to *define the static structure*. For that static structure we have *components* of different kinds (functional or data abstraction) and of different size (modules or subsystems). Components have interfaces according to kind and size. Between components we have different *structure* relations (local to, is a), and also different *import relations* (local, general, and inheritance import). Consistency conditions define the context sensitive syntax rules. There are *method rules* and *patterns* of different kinds giving hints how to use the language. The pattern discussion is broader than usual /BM 96, GH 95, SS 00/, as here different architectures (fig. 2), changes within a specific form and changes from form to form are involved. The presentation is *graphical* for surveys and *textual* for details.

Further notations are necessary, for defining the *semantics*, for *concurrency*, for *distribution, deployment*, etc. It can be shown that the notations for architectural *styles* /SG 96/ can be expressed by the aforementioned language concepts /Na 21c/. Therefore, these style notations can be used as abbreviations without escaping from the architecture language.

In /Na 21a/ it was shown that the above language and methodology *approach* is *integrative* in its nature. Many of the different and necessary parts for expressing aspects and views are found within one architecture language. The others are expressed by annotations inside of artifacts written in the language. It is always clear what these annotations belong to.

7 Summary

The output of a design and realization process for a software system is a *complex configuration* containing different abstractions, views, enrichments, which all deliver basic information and details. We find also different hierarchies within artifacts and between artifacts /Na 21a/. In this paper, we *concentrated* on the *architecture part*, which, although being a part of the overall configuration, has the same characteristics as the overall configuration.

That architectural configuration contains *more than one architectural description*. We have shown that the descriptions should be organized in different stages and that the order of these stages must be planned carefully. The order reflects abstraction, probability of changes, minimization of backtracking steps, and alike.

In this paper, we have discussed *two examples*, one from the domain business administration and one from the domain embedded systems. In this sense, this paper is the application of /Na 21a/, where we introduced the integrative approach for architecture modeling, as the architecture sequence for reflecting different aspects is one of the integration dimensions which need different notations which are also integrated. This means that different aspects of an architecture have to be denoted and that the mutual relations of these parts have to be obvious. We call this *integration* within an *architectural language and methodology*.

We also gave some *rules* in order to find the order in the architecture sequences. The right *order supports* the *adaptability* of the system and the *consistency* of the different architectural descriptions.

Further problems, which also have to do with different aspects of architectures were not addressed in this paper: as (i) *families of systems*, where we have to master their commonalities and differences, (ii) libraries of *reusable components* and their organization, or (iii) organizing *reuse* in building successively systems of a *certain application domain or structure*.

8 References

/Ar 10/ I. Armac: Personalized eHome Systems (in German), Doct. Diss. RWTH Aachen University , 315 pp., (2010)

/AU 20/ AUTOSAR: Automotive Open System Architecture, http://www.autosar.org/ (2020)

/BF 95/ P. Buhr/ M. Fortier et al.: Monitor classification, ACM Computing Surveys 27 (1): 63–107 (1995)

/BM 96/ F. Buschmann/ R. Meunier et al.: Pattern-oriented Software Architecture- A System of Patterns, Wiley (1996):

/Bö 94/ J. Börstler: Programming in the large: Languages, Tools, Reuse (in German), Doct. Diss. RWTH Aachen University, 205 pp. (1994)

/BR 05/ G. Booch/ J. Rumbaugh et al.: The Unified Modeling Language User Guide, Addison Wesley (2005)

/COR 20/ https://www.omg.org/spec/CORBA/About-CORBA/ (2020)

/Cr 00/ K. Cremer: Graph-based Tools for Reverse and Reengineering (in German), Doct. Diss., RWTH Aachen University, 220 pp. (2000)

/ET 08/ C. Eastman/ P. Teicholz/ R. Sacks/ K. Liston: BIM Handbook. John Wiley & Sons, 2008

/GH 95/ E. Gamma/ R. Helm/ R. Johnson/J. Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software, 395 pp. Addison Wesley (1995)

/GS 94/ D. Garlan/ M. Shaw: An Introductio to Software Architectures, TR CMU-CS-94-166 (1994)

/Gu 76/ J. Guttag: Abstract Data Types and the Development of Data Structures, Comm. ACM 20, 6, 396-404 (1977)

/Ja 00/ M. Jazayeri et al. (eds.): Software Architecture for Product Families, Addison Wesley (2000)

/Ki 05/ M. Kirchhof: Integrated Low Cost eHome Systems – Processes and Infrastructures (in German), Doct. Diss. RWTH Aachen University, 331 pp. (2005)

/Kl 00/ P. Klein: Architecture Modelling of Distributed and Concurrent Software Systems Doct. Diss. RWTH Aachen University, 237pp. (2000)

/KN 07/ B. Kraft/ M. Nagl: Visual Knowledge Specification for Conceptual Design: Definition and Tools Suppert, Journ. Adv. Engg. Informatics 21, 1, 67-83 (2007)

/Kr 07/ B. Kraft: Semantical support for the conceptual design of buildings (in German), Doct. Diss., RWTH Aachen University, 381 pp. (2007)

/Le 88/ C. Lewerentz: Concepts and tools for the interactive design of large software systems (in German), Doct. Diss., 179 pp. RWTH Aachen University, Informatik-Fachberichte 194,Springer (1988)

/LZ 74/ B. Liskow/ S. Zilles: Specification Techniques for Data Abstractions, Int. Conf. on Reliable Software, 72-87, IEEE (1975)

/Ma 05/ A. Marburger: Reverse Engineering of Complex Legacy Telecommunication Systems, Doct. Diss., RWTH Aachen University, 418 pp. (2005)

/Me 12/ C. Mengi: Automotive Software – Processes, Models, and Variability (in German), Doct. Diss., RWTH Aachen University, 350 pp. (2012)

/Mo 09/ Ch. Mosler: Graph-based Reengineering of Telecommunication Systems (in German), Doct. Diss. RWTH Aachen University, 268 pp., RWTH Aachen University (2009)

/Na 90-20/ M. Nagl: Software Engineering- Methodological Programming in the Large (in German), 387 pp., Springer (1990), plus further extensions over the time for a lecture on Software Architectures from 1990 to 2020

/Na 99/ M. Nagl (Ed.): Building Tightly Integrated Software Development Environments - The IP-SEN Approach, LNCS 1170, 709 pp., Springer, Berlin Heidelberg (1999)

/Na 21a/ M. Nagl: An Integrative Approach for Software Architectures, 25 pp. Techn. Rep. AIB 2021-02, Dept. of Computer Science, RWTH Aachen University

/Na 21b/ M. Nagl: Embedded Systems: Simple Rules to Improve Adaptability, 23 pp., Techn. Rep. AIB 2021-03, Dept. of Computer Science, RWTH Aachen University

/Na 21c/ M. Nagl: Architectural Styles in One Notation, forthcoming paper (2021)

/NM 08/ M. Nagl/ W. Marquardt: Collaborative and Distributed Chemical Engineering – From Understanding to Substantial Design Process Support, IMPROVE, LNCS 4970, 851 pp., Springer (2008)

/No 07/ U. Norbisrath: Configuring eHome Systems (in German), Doct. Diss. RWTH Aachen, 286 pp. (2007)

/NW 99/ M. Nagl/ B. Westfechtel (Eds.): Integration of Development Systems in Engineering Applications – Substantial Improvement of Development Processes (in German), 440 pp., Springer (1999)

/PB 05/ K. Pohl/ G. Böckle et al.: Software Product Line Engineering, 467 pp., Springer (2005)

/PD 07/ D. Pollet/ S. Ducasse et al.: Towards a Process-Oriented Software Architecture Reconstruction Taxonomy, Proc. 11th European CSMR 07, 137-148 (2007)

/Re 10/ D. Retkowitz: Software Support for Adaptive eHome Systems (in German), Doct. Diss. RWTH Aachen University, 354 pp. (2010)

/Ru 11/ B. Rumpe: Modelling with UML (in German), 2nd ed., 293 pp., Springer (2012)

/Ru 12/ B. Rumpe: Agile Modeling with UML (in German), 2nd ed., 372 pp. Springer (2012)

/SAD/ Software Architecture and Design Tutorial, TutorialRide.com, https://www.tutorialride.com/software-architecture-and-design/software-architecture-and-design-tutorial.htm

/Sc 13/ R.F. Schmidt: Software Engineering – Architecture-driven Software Development, 376 pp. Elsevier (2013)

/SEI 10/ Software Engineering Institute of CMU: What Is Your Definition of Software Architecture, https://resources.sei.cmu.edu/library/asset-view.cfm?assetID=513807

/SG 96/ M. Shaw/ D. Garlan: Software architecture: perspectives on an emerging discipline. Prentice Hall (1996)

/SS 00/ D. Schmidt/ M. Stal et al.: Pattern-oriented Software Architectures, vol 2 Patterns for Concurrent and Networked Objects, Wiley (2000)

/Ta 12/ A.S. Tanenbaum: Computernetzwerke, 5th ed., Pearson (2012)

Prof. Dr.-Ing Dr. h.c. Manfred Nagl, Emeritus
Informatics Department, RWTH Aachen University
nagl@cs.rwth-aachen.de

Aachener Informatik-Berichte

This list contains all technical reports published during the past three years. A complete list of (more than 570) reports dating back to 1987 is available from

`http://aib.informatik.rwth-aachen.de/`

or can be downloaded directly via

`http://aib.informatik.rwth-aachen.de/tex-files/berichte.pdf`

To obtain copies please consult the above URL or send your request to:

Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen,
Email: `biblio@informatik.rwth-aachen.de`

2018-02	Jens Deussen, Viktor Mosenkis, and Uwe Naumann: Ansatz zur variantenreichen und modellbasierten Entwicklung von eingebetteten Systemen unter Berücksichtigung regelungs- und softwaretechnischer Anforderungen
2018-03	Igor Kalkov: A Real-time Capable, Open-Source-based Platform for Off-the-Shelf Embedded Devices
2018-04	Andreas Ganser: Operation-Based Model Recommenders
2018-05	Matthias Terber: Real-World Deployment and Evaluation of Synchronous Programming in Reactive Embedded Systems
2018-06	Christian Hensel: The Probabilistic Model Checker Storm - Symbolic Methods for Probabilistic Model Checking
2019-02	Tim Felix Lange: IC3 Software Model Checking
2019-03	Sebastian Patrick Grobosch: Formale Methoden für die Entwicklung von eingebetteter Software in kleinen und mittleren Unternehmen
2019-05	Florian Göbe: Runtime Supervision of PLC Programs Using Discrete-Event Systems
2020-02	Jens Christoph Bürger, Hendrik Kausch, Deni Raco, Jan Oliver Ringert, Bernhard Rumpe, Sebastian Stüber, and Marc Wiartalla: Towards an Isabelle Theory for distributed, interactive systems - the untimed case
2020-03	John F. Schommer: Adaptierung des Zeitverhaltens nicht-echtzeitfähiger Software für den Einsatz in zeitheterogenen Netzwerken
2020-04	Gereon Kremer: Cylindrical Algebraic Decomposition for Nonlinear Arithmetic Problems
2020-05	Imke Drave, Oliver Kautz, Judith Michael, and Bernhard Rumpe: Pre-Study on the Usefulness of Difference Operators for Modeling Languages in Software Development
2021-01	Mathias Obster: Unterstützung der SPS-Programmierung durch Statische Analyse während der Programmeingabe
2021-02	Manfred Nagl: An Integrative Approach for Software Architectures
2021-03	Manfred Nagl: Sequences of Software Architectures
2021-04	Manfred Nagl: Embedded Systems: Simple Rules to Improve Adaptability

2021-05	Manfred Nagl: Process Interaction Diagrams are more than Process Chains or Transport Networks
2021-06	Manfred Nagl: Characterization of Shallow and Deep Reuse