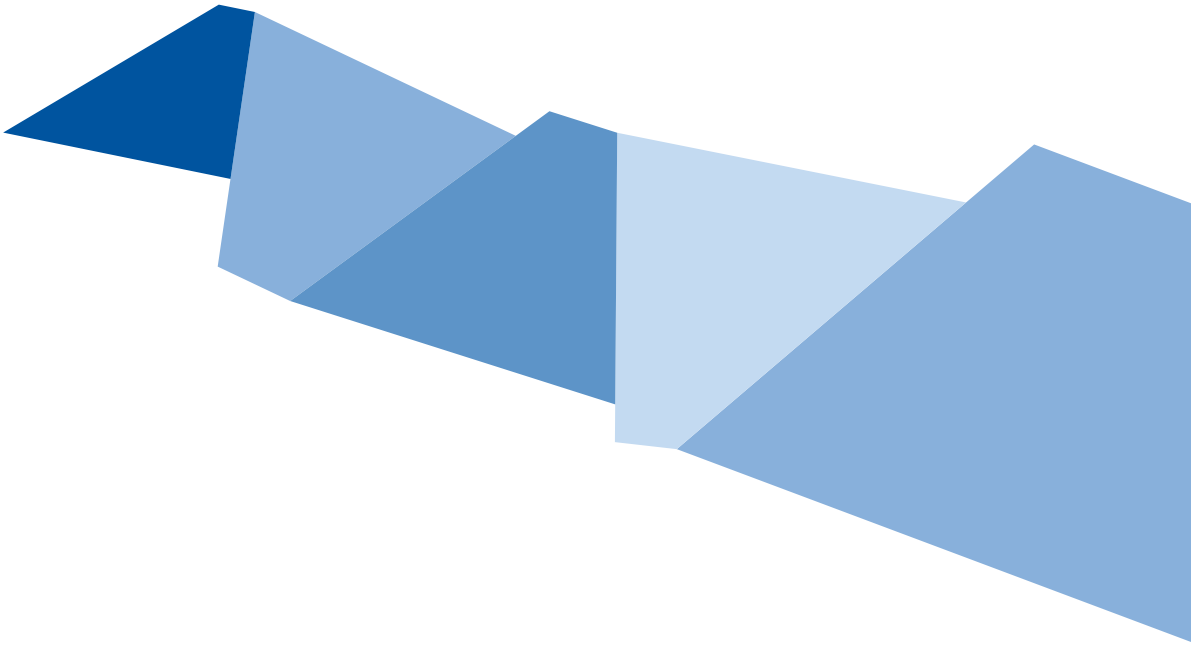


Characterization of Shallow and Deep Reuse

Manfred Nagl

[Nag21e] M. Nagl:
Characterization of Shallow and Deep Reuse.
In: RWTH Aachen University, Technical Report. AIB-2021-06. Feb. 2021.
www.se-rwth.de/publications/

Department of Computer Science

Technical Report

Aachener Informatik-Berichte (AIB) | ISSN 0935-3232 | AIB-2021-06
RWTH Aachen University | Department of Computer Science | February 2021

The publications of the Department of Computer Science of *RWTH Aachen University* are in general accessible through the World Wide Web.

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

Characterization of Shallow and Deep Reuse

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

Abstract
Reuse in software development is not only to carry out the same processes over and over again, thereby being more efficient, i.e. faster and producing less errors. We call these forms *shallow* reuse, where reuse is mostly in the mind of developers. *Deep reuse* means to change the development process remarkably, because upcoming knowledge makes more or less big parts of the development superfluous. Examples are that components and frameworks from former developments are used, steps are automated, etc.

In this article, we try to clarify the difference of shallow and deep reuse. Furthermore, we *characterize the changes* due to reuse on three levels: the new product with an improved form of reuse, the change of the development process, and the new parts to be reused in the future. A new notation for processes makes the changes and the dependences of subprocesses more evident.

We take the *multiphase compiler* as the running example, as it is one of the best studied software products, a good example for the combination of theory and practice, and also of deep reuse forms.

Key words: implicit (shallow) and explicit (deep) reuse, elaborated reuse forms as table driven software approach or generation approach, intelligent architectures, global architecture patterns, multiphase compilers and compiler compiler approach, reuse steps: changes of product and process

1 Introduction

Shallow reuse means to repeat the development, thereby shortening the time of development and increasing its quality. The corresponding steps of improvement are characterized by models like Capability Maturity Model (abbr. CMM) /PC 95/, namely to use documentation, to manage, and to optimize the process. Doing a similar software development task again can speed up the productivity. Many other factors also influence productivity and quality /SZ 19/. This all is important, but a development process having all these characterizations does not change its internal structure radically.

In shallow reuse, *reuse* is *implicit*. The corresponding reuse knowledge is in the minds of developers, as the way to proceed, an imagination of the form of the result, the experience gained from former development processes of a similar system, etc. Reuse mostly makes use of copy, paste, and further changes. There is mostly *no explicit* and important *result* left for influencing the following development processes. Especially, there is no clear improvement of development knowledge.

This form of reuse is the *classical* one, and still is the *mostly applied* approach in companies and *industry*, even if developers are working for a long time in one specific domain, thereby producing many similar systems. This form of reuse can be called historical, as to be seen from a characterization of the design of Gothic cathedrals (12^{th} to 15^{th} century) and the classification of reuse forms /Na 19/. There, we already find most of these reuse forms, although there was no easy to use and mechanical way for copy, paste, or change at that time.

In this paper, we go further. *Deep reuse* means to learn from one development for the next by facilitating and automating the development process. This means that reuse results are made *explicit*. This can be easily seen from the changes of the final result and the corresponding process transformations. The aim of this paper is to clarify these changes and transformations from step to step and to show the tight relations between product and process modifications.

For this clarification, we use one of the best examples of intelligent development – the *construction of a multiphase compiler*. There are also other examples of *intelligent development*, which could have been used, as (i) the development of communication protocols by specification and code generation /GG 07/, (ii) the development of control and automation software by making use of a predefined set of partial solutions which are graphically composed /WK 16/, or (iii) the development of novel tools for software construction in the academic context (/Na 96/ and also for other engineering domains).

In the following paper, we use a *compact product description*. As the architecture of a software system is a master document for the whole development process /Na 90-20, Na 03/ we use an architectural notation, here in a compact form /Na 21a/, which is influenced by different programming languages /Wi 21b/. For the development processes we use a notation /Na 21d/ which specifies the dependency relations (output of a process – input in the next process) between processes by regarding the purpose of use in the second process. They are called *Process Interaction Diagrams* (PIDs). That makes these dependency relations more specific.

The example regarded in this paper contains *shallow reuse* forms at the beginning, before the knowledge about reuse is explicitly extracted and applied to form new reuse steps. There, the product is changed due to invariant components and frameworks, and the process is partially automated (by the generation of software, table-driven approaches, or even generation of tables). These steps we call *deep reuse*.

Nothing of the reuse steps explained in this paper is new. However, the paper delivers a *new characterization of reuse*. Especially, it makes clear that deep reuse is what we should look for when building systems for a long time. The paper is also a plea for deep reuse in industry, as such deep reuse is mostly applied only in academic ecosystems.

This paper is *not a survey* on the broad field of *software reuse*, or of the different aspects of reuse, or of the vast amount of literature on software reuse. For a first view see /Co 98, LS 09, ICSR/. Instead, this article takes just one interesting example and characterizes two forms of reuse (what is usually done vs what could have been done), by giving interesting characterizations of the product and the process in the second case in form of different steps of reuse.

2 The Example: A Multiphase Compiler

We assume that the programming language to be compiled belongs to the class of clearly-defined, strongly-typed languages, with many forms of syntactical checks at compile time. *Compilation* by a multiphase compiler (see e.g. /AS 06, Sc 75, WG 84/) is *done in* steps (*phases*), see fig. 1 for a corresponding sketch of an architecture diagram: scanning, parsing, context sensitive analysis (also called static semantics), intermediate code generation, optimization, addressing / target code generation, and post optimization.

All these steps are represented as *functional components*, more likely as composed subsystems than atomic modules. The underlying structure is *functional composition*: splitting a complex function component into simpler functional components executed one after the other, using intermediate data structures, see again fig. 1.

Scanning is analysis on the lowest level of syntax, reading text input and building up lexical units (tokens), as identifiers, word symbols, delimiters, or literals. *Parsing* means to analyze token sequences whether they reflect the context-free syntax, e.g. an assignment consists of a name on the left hand side, an assignment symbol, an expression on the right hand side, and a semicolon at the end. *Context sensitive analysis* checks for consistency of parts, which may be far away from each other, as declaration and application, and so on. The left part including intermediate code generation is called the *front-end* (being programming language dependent), the right part is called the *back-end* (being target machine dependent).

Fig. 1.a reflects the *structure* of the compiler after *several iterations*. From version to version the compiler structure became clearer, the compiler development was more efficient in development time and quality (less number of errors/ mistakes), and the compiler runtime as well as the runtime of compiled code was improved. Possibly, the compiler was developed in two versions, as a students' compiler for fast compilation and an optimized compiler for delivering runtime-efficient code. The compiler structure was possibly applied for compilers of different languages, for different versions of the same language, or for different target machines in a software house specialized on compilers.

Compilation of a programming language is a specific and clearly describable problem, to a big part formally. It was studied for a long time in the 60ies to 80ies /Wi 21a/ by some of the brightest persons of Computer Science. Underlying there is a *specific problem*: *precise description* of input (language), output (machine), and of every internal part. There are no vague parts like "The system must have a nice user interface", "the system must use that and that given big component or technology", "the systems must run in the following specialized target environment", "the system must fulfill that hard efficiency requirements parameters", etc. The knowledge how to write a compiler grew over time, from well-understood handwriting /Sc 75, Wi 77/to refined forms in later times using automatisms, see below.

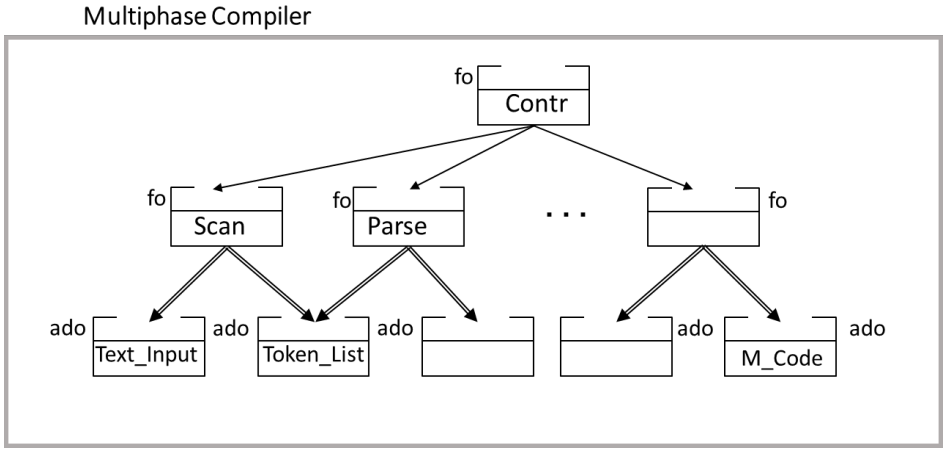
That is why we find clear solutions and a theory on which solutions are based /AS 06, WG 84/. Compiler writing is one of the best examples we have in Informatics for *combining theory and practice*. Of course, not every problem, to be solved by a software system, has the above nice features. However, when building similar systems for a certain application domain

and for a long time, you will always detect different forms of reuse, of classical reuse and of deep reuse, if you have time, money and good people. Quite probably, not the whole program system might make use of intelligent reuse forms. But you find them at least in certain parts of the system.

The *underlying data structures* are (i) the list of lexical units (tokens) for the input text streams produced by Scan, (ii) the abstract syntax tree, produced by Parse, the program graph (syntax tree together with symbol list, or attributed syntax tree), and so on.

Fig. 1.b contains the *process to develop the compiler* according to fig. 1.a. We assume that the language L and the target machine M are determined.
In the requirements part we collect information for the language, the target machine, and corresponding compiler techniques. This information is passed forward.
Then, the Realization process starts. All of the output info of Collect_Info goes to Build_Arch. There, we design the compiler, see again fig. 1.a: the control component, the different functional parts and the data structure components of the bottom, and see corresponding processes, for design and realization. The corresponding information of the language (grammars) is forwarded as constraints to the corresponding phases of the front-end. The lexical syntax goes to Scan, the context-free syntax to Parse, etc. The information corresponding to compiler techniques is forwarded to all phases and, especially, to basic_Layer. The information corresponding to the target machine is forwarded to the back-end components.
We do not regard the internal structure of the subsystem components, by reasons of simplicity. All the components identified in the architecture, have to be *implemented*. Afterwards, the component Int *integrates* the results.

Fig. 1.c delivers the *captains* for fig. 1. We denote the different aspects of the processes (fig. 1.c right), which allows to make the dependency relations between processes to be more precise. Fig. 1.c. left explains the notations used in the architecture.



a)

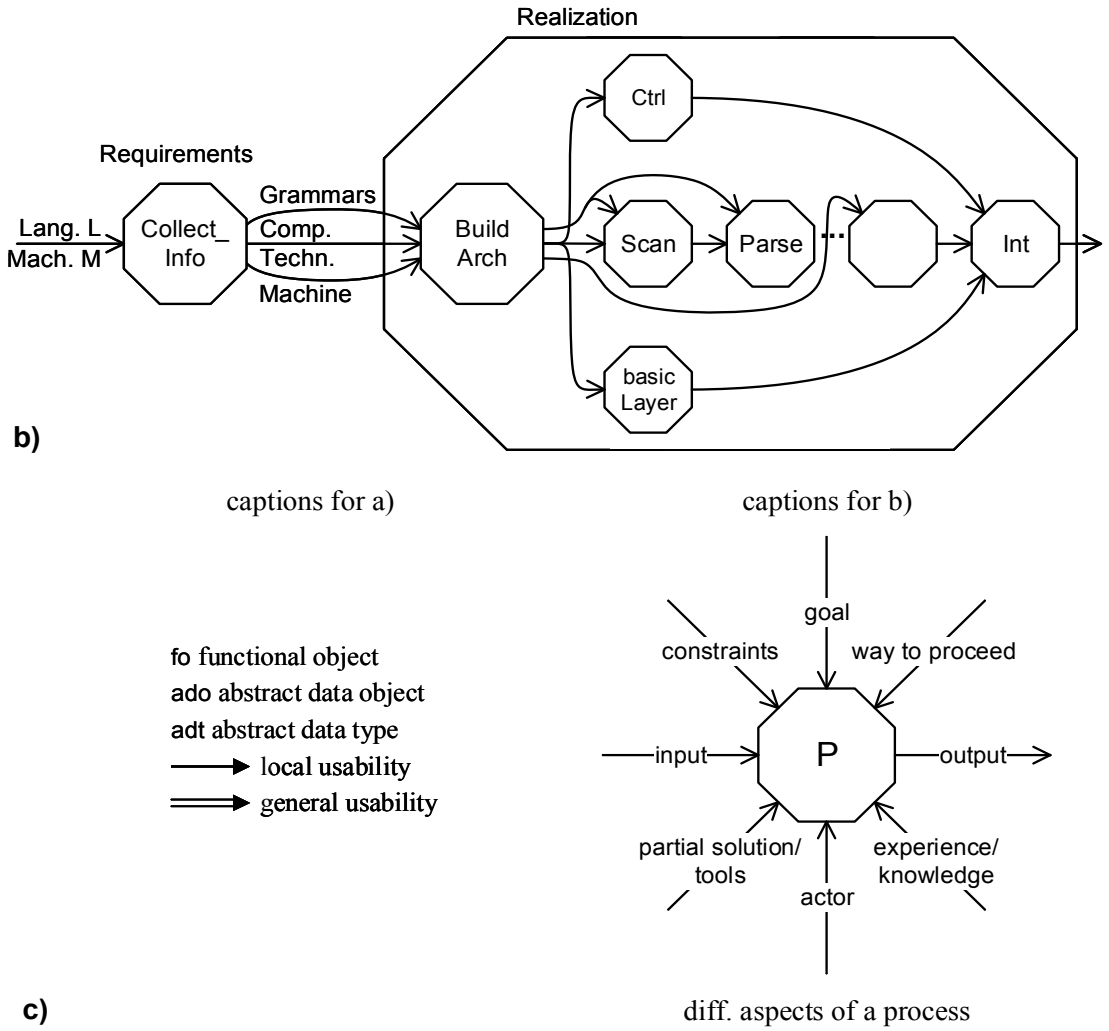


Fig. 1: a) The architectural structure of a multiphase compiler and b) its development process (after several iterations), c) captains for parts a) and b)

To explain the *notational elements* of the architecture in fig. 1.a: fo stands for an object component of functional nature, ado for an abstract data object, or adt (abstract data type) in case of separate compilation, i.e. more than one compilation unit can be compiled in one compilation run. Local (thin) and general (double lines) usability indicate specific or general usability. The components of the front-end are language specific, those of the back-end machine specific.

We *denote* elementary *processes* of fig. 1.b by an octagon, hierarchical processes by a net inside an octagon. The dependency of subprocesses is specific, see the following explanations and /Na 21d/.

3 The Compiler after Detection of Standard Structures

Fig. 1 is the last form of implicit (shallow) reuse, having the corresponding knowledge only in mind. However, it is also the *beginning of explicit* (deep) *reuse*, as indicated in fig. 2.a by different colors.

The global structure of the compiler (in blue) is invariant, it is a global build plan or a *global design pattern*, going here down only to the level of subsystems. The control component Contr (green) is invariant and can be used in every compiler, the functional components (black) are internally different, from language to language.

Even more, at the bottom of the architecture diagram of fig. 1, we see data exchange components (again green), which form a *reusable basic layer of data structures*. This layer can immediately be reused in the next compiler project, as long as the functional structure of the compiler remains the same. For the basic layer, we have a series of implemented and ready to use components, which can be reused. Internally, they should be *adaptable*, to be connected to the underlying and specific file systems of the development environments. Again, the components are more subsystems than modules (so-called entry-collection subsystems /Na 21a/).

Putting together: The diagram of fig. 1 contains a *standard structure* for compilers of the above language class, a framework as a *global pattern* (in blue), where we find reusable components (in green). For the components of the phases we have to deliver specific components (design and implementation, in black). The components in green for control and for the basic layer of data structures can *directly be used* for the next compiler, they are already implemented. So, we have a plan to reuse (framework) and also different components, which are ready to be reused. The *basic data structures* for *data exchange* have a universal form for multiphase compilers.

Now, we start the development of the next compiler and reuse what we have identified, see fig. 2.a. The remaining parts to be developed are highlighted in black, the parts already carried out are drawn in green. Only the phase components have to be designed and later implemented, the rest is already done (components Ctrl and basic layer for data structures). The global build plan is the same.

The development process, shown in fig. 2.b, has also changed remarkably .The process is much simpler now. If the language is new, the Collect_Info part has to be done again, mainly to pass the corresponding knowledge. If the language is the same (we only want to restructure the compiler to make use of reuse), these components' results can just be taken. Only the processes for developing the phases remain, presented in black, the other parts already available are drawn in green. Furthermore, the integration Int has to be done again.

The upper parts of figs. 1 and 2 show the *change* of the development *product*, the lower parts of both figures show the change of the *development process*. The green parts say what we get from the last and previous developments, either *as product or as process*.

For reasons of simplicity, we concentrate in the following on a apart of the front-end in the following explanation. We *take the parsing as example*. A similar argumentation holds, if we would have taken scanning, context sensitive syntax analysis, or intermediate code generation, assuming we have the same degree of knowledge, see below. Parsing is the phase best studied in literature.

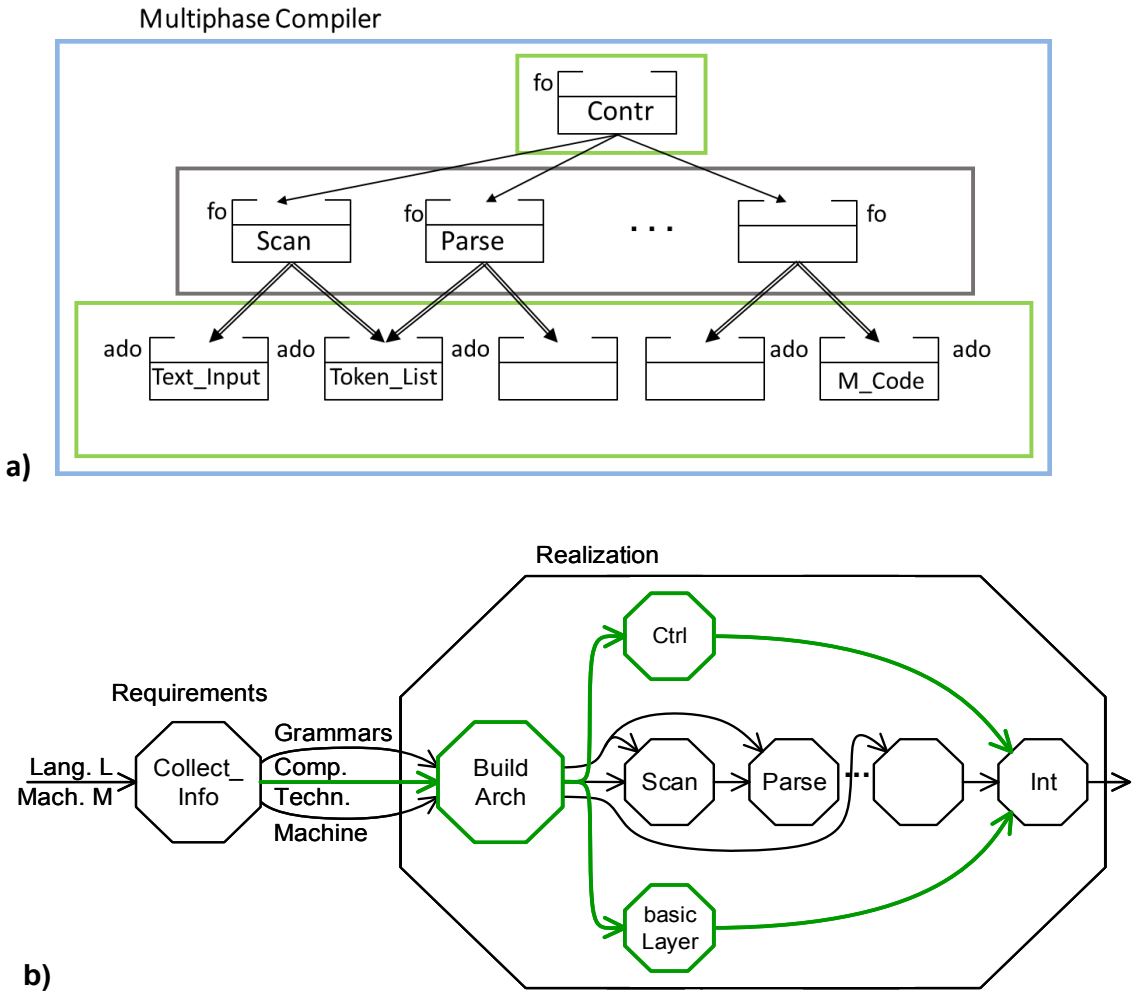


Fig. 2: The next iteration: a) Standard for the product as framework, reusable Contr, basic bottom-layer for data structures, b) realization of the next compiler: corresponding processes

4 Deep Reuse: Table-driven Approach

The typical *deep reuse steps* are explained in the next three sections. For the next two sections we take the parser as the running example. Thus, we regard only a cutout of the architecture. We assume that the corresponding reuse steps can be taken for all phases of the front-end.

As example, we look at the hardwired functional component for parsing of above. In the table-driven approach, the *table* describes the *behavior* of the parser. We need an additional component, called *driver* or interpreter for handling parsing by use of the table. Both together have the same functional behavior as the hardwired program component.

So, the *necessary steps* are: (a) The driver component development, which is only implemented once for the regarded parsing method (e.g. LL(1) or LALR(1)). Therefore, we show the component of the architecture as well as the corresponding subprocess in light brown. They can be reused immediately afterwards. Now, (b) we build up the table according to the deterministic context-free grammar. This component and also the subprocess are drawn in black.

The *steps in direction of reuse and development efficiency* are: (i) Driver only once then arbitrarily reused. (ii) to build up the table is easier than programming the parser, see fig.3. The remaining *development step* to build up the table. The same approach can be applied for the scanner. There, we take a regular grammar or a finite automaton as formal description.

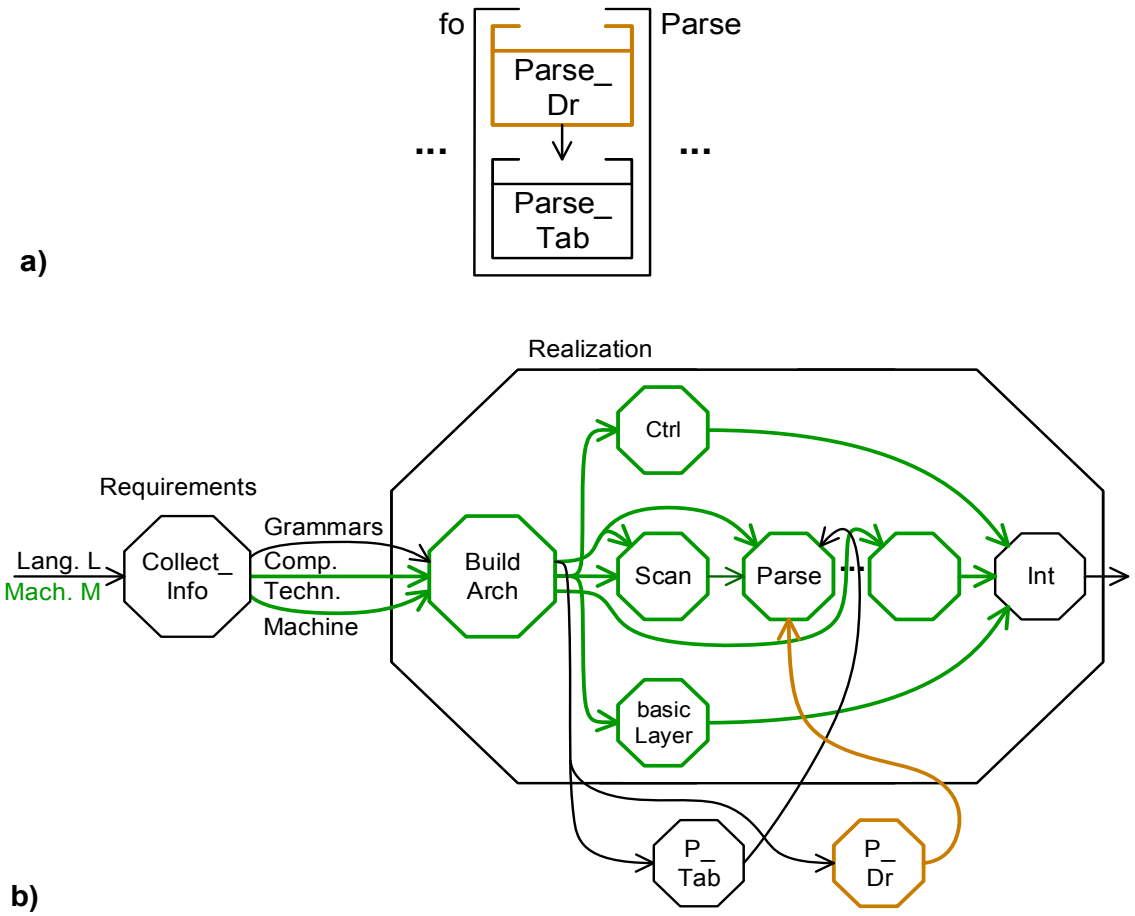


Fig. 3: a) Parse is now different, see cutout of the architecture, b) Table-driven architecture, process: driver (only once) and table to be developed

5 Deep Reuse: Generation of solutions

In fig. 2.a the *code* for Parse was *developed by a human*. This person looked on the grammar and transformed this grammar into a component, being an analyzer for this grammar. Analogously, in fig. 3 the *table*, describing the behavior of the analyzer, was produced by a human.

The tasks to transform grammar either to code or to a table are *precise* and can be *formalized.* We can solve this *transformation by a program*. The corresponding approach is called to generate the result, or automation by *generation*. The program, which is necessary, is universal for the type of grammar. Thus, it has to be developed only once, and it can be used repeatedly. In fig. 4.b the corresponding automating process, therefore, is again painted in light brown.

In fig. 4.a we see the *resulting cutout of the architecture.* The Par_Dr is reused (green), the table Par_Tab is generated now, also green. In both cases, there is no development process.

The development of the generator has to be done only once, so it is again painted in light brown.

Fig. 4.b shows the situation generating the *parser table*. The part already done is shown in green. The corresponding driver is available, as having been developed, see fig. 3. The corresponding table is produced automatically by the generator. The generator is already available. So, if a corresponding grammar is available, nothing has to be done. Parse is no longer a development process, it is an automatic process.

If the *parser code is generated*, fig. 4.b is simpler. No driver component is necessary. Again, if a corresponding grammar is available, nothing has to be done by a developer, as the parser code is generated. The generator (grammar to code) has to be done only once (light brown).

What we have discussed for the parser can also be done for the scanner, here a regular grammar or a corresponding finite automaton is necessary. More generally, it can be done for any part of the compiler, which can precisely be described. Such parts of *automatable transformations* are more in the compiler front-end than in its back-end. We find such problems also *outside of compiler construction*. Then, if the generator is available, we only have to look for a corresponding formal input description.

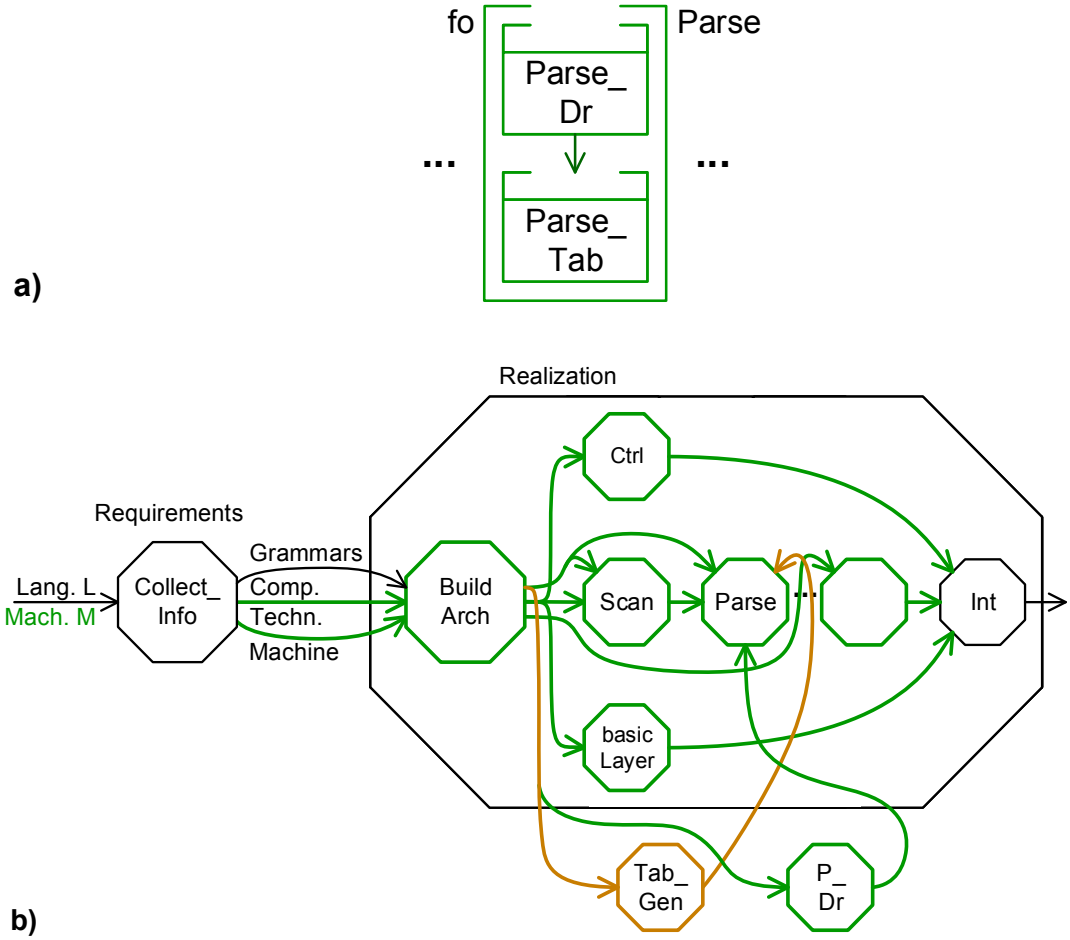


Fig. 4: Automation: Generate code or table, here for the example parsing

Summing up sections 4 and 5, we have done another *big step in direction of reuse*. In the table case, (i) the drivers are reusable, after having been developed. In both cases, generated table and generated analyzer program, (ii) the generators are reusable, after having been developed. Furthermore, *automation* not only reduces the *effort*, it (iii) even *eliminates* the corresponding development by a human.

In the discussion above, we denoted *processes* or *components*, where nothing is to do, as a previous process delivered a product which can be reused, in a green color. The green color shows what has already been done, and what is not necessary to be done again. A green process is *trivial*: The program (component) is available, a driver with a table is available, the program or the table has been generated, etc. Therefore, the process just means to take something out of a library and use it. As an example, look on process `Parse` of fig. 4.We did not introduce trivial processes as distinction to usual processes for characterizing this situation.

Above, we saw *sequences of architectures*, the transitions from architecture to architecture being defined by a deep reuse step. All these architectures are abstract, i.e. they do not refer to technical details of the system to be developed or its environment. Specifically interesting but also nontrivial is reuse for embedded systems, as these systems are connected to technical details. There, we also have sequences of architectures, but there to transform an architecture from an abstract to a concrete form, reflecting the technical details of the system or its environment /Na 21b/.

6 Deep Reuse: Global Reuse Schemes

The summary of all from above (framework, basic layer, table-driven components, the generation of components) is called the *compiler compiler* approach. It is one of the best examples of deep reuse and also of a close connection of theory and practice.

It is *deep reuse*, as in every step of figs. 2, 3, and 4 the development product and its process were *drastically changed*. This reorganization demanded deep understanding (extracting the global build plan, extracting reusable control and basic layer, avoiding to program by using a table with a driver, avoiding to program or to develop the table by automation through a generator. Deep reuse does not mean to do it only better again, but to learn and to *reorganize* the product and the process essentially.

The discussion of above also gives rise to study global reuse patterns. The first is *front-end reuse*. Here, we use the compiler front-end and care about different back-ends for various target machines. In this case, the same front-end (for the same language) is used for different back-ends (different target machines). The different back-ends have to be developed.

Analogously *back-end reuse* means to have different front-ends to be developed (languages, here compiler-oriented). Here, the back-end is stable (we have the same target machine).

A combination of both reuse approaches requires a uniform intermediate language (e.g. graph intermediate language), see fig. 5. This is combination called the *UNCOL* (Universal Computer Oriented (Intermediate) Language) *approach* /Co 58/, an old and still very important idea, which can be used for any translation problem.

Using UNCOL *reduces* the *effort* and thus is an important and global *reuse method*: If we directly translate n languages and want to have the compiler on m machines, we have a development effort for n*m compilers. Using a uniform intermediate code the effort is reduced to n+m, a dramatic reduction. Of course, the idea only works for similar translations, in our case compiling typed, compiler-oriented languages with a fixed multiphase scheme, and an agreement for a universal intermediate code.

Above, we mostly had development *product reuse*: global pattern, basic layer, table and driver, altogether products of subprocesses, which can be used and reduce the development effort. Only the generator for code or tables belongs to *process reuse*, here in a mechanical form.

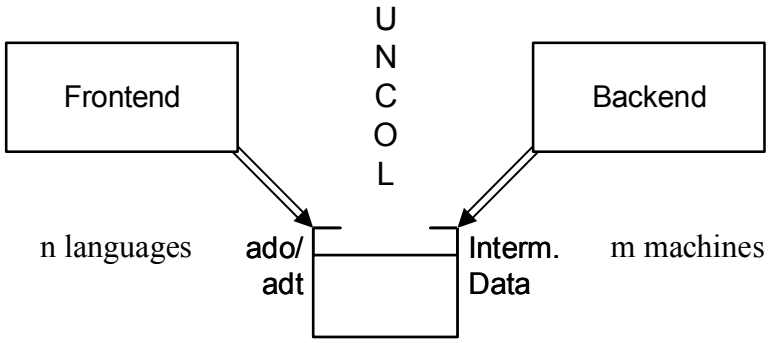


Fig. 5: The UNCOL approach as a widely used reuse approach: scheme and effort reduction

Recursive Descent compilers /Wi 77/ work differently, compared to multiphase compilers. Whereas in multiphase compilers the total task of translation is expressed by a sequence of subprocesses, in a recursive descent compiler we have only one pass working top-down (one pass compiler). In this pass, *all subtasks of above are solved for a language construct*. i.e. a declaration, an assignment, a control structure, or a program structuring construct like a procedure. The book /Wi 77/ nicely describes this rather easy way of compilation, such that a student can write a compiler for a simple language according to that scheme after a short time.

In this case, we have *process reuse*: A methodology is introduced which says what to do and in which order, such that you can *apply the methodology* for a compiler task without getting into hard problems. No intermediate product is reused, everything is done again (for a language, for a machine).But the methodology is reused, which says what and how to do. Therefore, we call this process reuse within the development process.

For languages and one pass compilers there is a method for getting a compiler called *bootstrapping*, which is a mixture of well-defined development subprocesses by a human and automatic subprocesses, This approach for a *stepweise compiler development* is also described in /Wi 77/. Nearly all Pascal compilers have been gained via this method. The method can be used for extending a programming language, for porting a compiler, for improving a compiler, etc.

The starting point is a *compiler for a language written in the same language* (e.g. Pascal compiler written in Pascal) and a one *available compiler* written in M for the machine M.

Then, after a sequence of human development steps and automatic compilation steps, we get the result. We give no explanation here and refer to the literature. The explanation for the example ‘extension of a language’ in the notation used in this article is given in /Na 21d/.

7 Further Examples of Deep Reuse

Further Examples

Above, we have used the multiphase compiler for classical programming languages as the example to demonstrate deep reuse and changes of the corresponding process. We could also have taken *tools* for the *software development process*. In the academic area, we find novel tools on different levels and also their *integration* in a tool environment (e.g. /Na 99, NM08/), which have been realized with deep reuse, quite similar to the compiler example.

There are further examples for such deep reuse. We find it in the area *automation and control* /HS 19, Po 94/, especially around the IEC 61131 and 61499 norms. There exist libraries of *predefined functions* for controllers and graphical tools to *arrange networks* of such controllers. Building a solution is often just a graphical task, no programming is necessary anymore.

Another area for advanced software reuse is *protocol* development for *telephone and computer Networks* /Ho 91, Sc 96/. There are solutions generated from a protocol specification. Here again, no usual programming takes place.

Further Approaches for Reuse

There is a branch of software engineering, dealing with *families of systems* /CN 07, Ja 00, PB 05, PK09/, not with single systems. Necessary for that is to detect the commonalities and the differences between members of the family. They have to be carefully studied and they are used for the implementation of the members. That is a big structural reuse advantage compared to program one family member after the other.

Another branch is *domain-driven development* /Ev 04/ which means to firstly develop solutions for the domain (knowledge, components, tools) and reuse them for the construction of examples of the domain. This intersects with *model-driven development* /PB 16, PH 12/, where the solution is described by models, from which a solution can be derived or in the best case generated automatically by a generator.

In all these approaches mentioned in this section, the early solutions started with hardwired programming for examples. Then, in these examples or in the corresponding approaches thinking began to avoid doing programming over and over again. The *process for* getting a *solution* was *changed dramatically*, and intelligent reuse approaches were used.

We could have taken one of the *above examples* or examples of the above *approaches* to demonstrate, what we have shown in this paper for compiler construction: There are different reuse steps, which change the product, the process, and which introduce further components, basic layers, or generators step by step, ending with an *intelligent and deep reuse process*.

8 Summary, Importance, and Open Problems

Summary and Lessons Learned

Above we have seen: Thinking in structures and reuse needs the right level, namely a *carefully designed software architecture*, which on one hard is precise and clear. On the other hand, it is adaptable, i.e. prepared for extensions and changes of functionality, but also for technical modifications like exchange of basic components as e.g. basic data structures. Functional and data abstraction have to be used. Careful discussions, design experience, and new ideas change the way of development.

Reuse is longtime learning: At the beginning we build several times, get experience, get more efficiency – in time and costs – from development to development. This was called in this paper classical or *shallow reuse*. Then, we make steps in direction of standardization and patterns, as frameworks and basic layers, so into deep reuse.

After further experiences and careful discussions *deep reuse* can continue. This, again, demands for intelligent people and especially for time. The result is a long-term advantage, which does not come for free, as shown in this paper for multiphase compilers of classical programming languages. This explains that deep reuse is rather seldom in industry, as it costs. The advantage, however, is a long-term profit.

Deep reuse in our in our compiler example *means*: detailed requirements, a careful architecture (fig. 1), using the right abstractions, furthermore looking for adaptability, for a framework, and a basic data exchange layer (again fig. 1). Afterwards, only the functional layer needs to be developed (fig. 2), the other parts remain from the last for future development. The *next steps* are tables for functions and corresponding drivers (fig. 3), generation of code or tables (fig. 4). Altogether, this is called the compiler-compiler approach. Global reuse patterns, like the UNCOL approach or bootstrapping, finalize the main sections.

That all did not come for nothing: Hard work was necessary and knowledge to see the long-term advantages. So, *deep reuse* methodology is only realistic for *stable domains and stable problems,* from the economy point of view.

If you look on the product (architecture) and the process, you can easily *distinguish between shallow and deep reuse*: As long as the product and the process remain similar, we have shallow reuse (fig. 1). If the product and the process change noticeably and in direction of reuse, we have deep reuse (figs. 2, 3, 4, 5).

Compilers are Important for the Informatics Education

The compiler is a special case in a specific class of systems, named *batch* systems. In this paper, the specific batch system is the *multiphase compilers* for classical languages. It has nice properties, as no vague and fuzzy aspects occur. The specific class uses functional decomposition and data abstraction structures for coupling the phases.

Why are *compilers* so *important for CS education*? There is a series of answers:
- We do not really *understand a programming language* if we do not see what happens in the

compilation process and what happens at runtime. The interesting thing of compilers is that they introduce the corresponding administration at compile time for the code, being executed at runtime.
-To say it in other words, we do not *understand to program*, if we do not have a basic knowledge of compilation and runtime systems.
- It is not very likely that CS students are later busy in a compiler company. This argument was used for *eliminating* compiler construction from many CS curricula. However, this is *not appropriate*.
- Furthermore, we quite often find *translation problems* in *practice*, where it is useful to have the corresponding background for solving these problems. The best preparation is to have a background in compilers.
- Compiler construction is the *parade discipline* for a close *connection* of *theory* and *practice*.
- *Deep reuse* demands for a careful discussion how to solve a problem. So, if you want to educate your students not only to hack, but to argue and learn, they should learn compiler construction. It would be nice and also important to have more areas, which combine theory and practice in a similar way.
- *Tools* need precise and formal languages, which demands for syntax, semantics, and pragmatics knowledge. Although most tools work iteratively and incrementally, the tool development process is similar to compiler construction. Compiler knowledge prepares for tool construction.
- Nearly all of the *brilliant ideas* to structure *software* and their *architecture* - and consequently also *reuse forms* - have been developed in compiler construction: framework for global patterns, universal structures for data exchange, table-driven approaches, generation approaches, bootstrapping, etc. So, missing compiler construction means to miss to learn these intelligent software construction principles and examples.

The Big Open Problem: Dissemination of Deep Reuse

We miss to have more intelligence in software construction. As stated above, the *simple reuse* forms (shallow reuse) are dominating in industrial software development. This is acceptable *in an ever changing* context: Informatics is developing rapidly, extensions of applications appear often, using new methods and tools happens frequently, starting new application domains is daily practice, etc. This hinders to think in the long term. Time is hectic – there is no time and money for sustainable development, and there is no corresponding understanding and insight at the management level of companies.

If however, you develop software more often in a *stable environment*, you should think about careful development and intelligent and deep reuse. That needs time, money, and excellent developers, but it is worth as *long-term investment*. Therefore, there is always a weighting long-term versus short-term profit. The long-term profit can be enormous. But it does not come for free.

Deep reuse only works for *specific classes* of *software systems*. The more *specialized* and precise the class is, and the more you *know* about this class, the *deeper* are the results. As already stated, you need intelligent developers and log-term thinking.

In this way, deep reuse contributes to the knowledge of software classes, to deep understanding in your company, and to deep understanding in software engineering subdomains. So, try to do and *contribute* to this way of *knowledge acquisition*.

As already argued, *families* of systems, *model-*, and *domain-driven* development are even more ambitious problems than single system development. In all these cases, we stay in a certain domain for a while and we develop different systems for a certain problem. Thus, the plea for *deep reuse* is even more important here.

9 References

/AS 06/ A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman: Compilers: Principles, Techniques, and Tools, Addison-Wesley, 2nd ed. (2006)

/CN 07/ P. Clements/ L.M. Northrop: Software Product Lines: Practices and Patterns, 563 pp., 6th ed., Addison Wesley (2007)

/Co 58/ M.E Conway: Proposal for an UNCOL, Communications of the ACM. 1 (10): 5–8 (1958)

/Co 98/ B. Coulange: Software Reuse, 293 p., Springer (1998)

/Ev 04/ E. Evans: Domain-driven Design, Addison Wesley, 529 p. (2004)

/GG 07/ R. Grammes/ R. Gotzhein: Fundamental Approaches to Software Engineering. Lecture Notes in Computer Science 4422, Springer. pp. 200–214 (2007)

/Ho 91/ G. J. Holzmann: Design and Validation of Computer Protocols, 539 p., Prentice Hall (1991)

/HS 19/ B. Heinrich/ W. Schneider: Grundlagen der Regelungstechnik, 5. Aufl., Springer Vieweg (2019)

/ICSR/ International Conference on Software Reuse, Proc. from 1990 to 2017, see also Wikipedia ICSR

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

/LS 09/ R. Land/ D. Sundmark et al.: Reuse with Software Components – A Survey of Industrial State of Practice, in Edwards/ Kulczycke (Eds.): ICSR 2009, LNCS 5791, 150-159 (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 IPSEN Approach, LNCS 1170, 709 pp., Springer, Berlin Heidelberg (1999)

/Na 03/ M. Nagl: Introduction to Ada (in German), 348 pp., Vieweg (1982), /Na 03/ 6th ed. Software Engineering and Ada (in German), 504 pp., Vieweg (2003)

/Na 19/ M. Nagl: Gothic Churches and Informatics (in German), 304 pp, Springer Vieweg (2019), see pp. 179-187

/Na 21a/ M. Nagl: An Integrative Approach for Software Architectures, Techn. Report AIB 2021-02, 26 pp., Computer Science Dpt., RWTH Aachen University (2021)

/Na 21b/ M. Nagl: Sequences of Software Architectures, Techn. Report AIB 2021-03, 16 pp., Computer Science Dept., RWTH Aachen University (2021)

/Na 21c/ M. Nagl: Embedded Systems: Simple Rules to Improve Adaptability, Techn. Report AIB 2021-04, 23 pp., Computer Science Dpt., RWTH Aachen University (2021)

/Na 21d/ M. Nagl: Process Interaction Diagrams are more than Process Chains and Transport Networks, Techn. Rep. AIB 2021-05, 18 pp., Computer Science Dpt., RWTH Aachen University (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)

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

/PB 16/ K. Pohl, M. Broy, M. Daemkes, H. Hönninger (Eds.): Adavanced Model-based Engineering of Embedded Systems – Extensions to the SPES 2020 Methodology, Springer, 303 pp. (2016)

/PC 95/ M.C. Paulk/ V.V. Weber/ B. Curtis/ M.B. Chrissis: The Capability Maturity Model: Guidelines for Improving the Software Process. SEI series in software engineering. Reading, Mass.: Addison-Wesley (1995)

/PH 12/ K. Pohl, K. Hönninger, H. Achatz, R. Broy (Eds.): Model-based Engineering of Embedded Systems – The SPES 2020 Methodology, Springer, 304 pp. (2012)

/PK 09/ A. Polzer/ S. Kowalewski/ G. Botterweck: Applying Software Product Line Techniques in Model-based Embedded Software Engineering, Proc. MOMPES'09, 2-10 (2009)

/Po 94/ M. Pohlke: Prozessleittechnik, 2nd edition, Oldenbourg (1994)

/Sc 75/ H.J. Schneider: Compiler, Aufbau und Arbeitsweise, de Gruyter (1975)

/Sc 96/ M. Schwartz: Broadband Integrated Networks, Prentice Hall (1996)

/SZ 19/ C. Sadowski/ T. Zimmermann (Eds.): Rethinking Productivity in Software Engineering, Springer Science+Business Media (2019)

/WG 84/ W. M. Waite/ M. Goos: Compiler Construction, Springer (1984)

/Wi 77/ N. Wirth: Compilerbau, Grundlagen und Techniken des Compilerbaus, Oldenbourg (1997), first ed. 1977

/Wi 21a/ Wikipedia: History of compiler construction, access Jan 2021

/Wi 21b/ Wikipedia: History of programming languages, access Jan. 2021

/WK 16/ C. Wagner/ D. Kampert/ A. Schüller/ F. Palm/ S. Grüner/ U. Epple: Model based synthesis of automation functionality, at – Automatisierungstechnik; 64(3), 168–185 (2016)

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