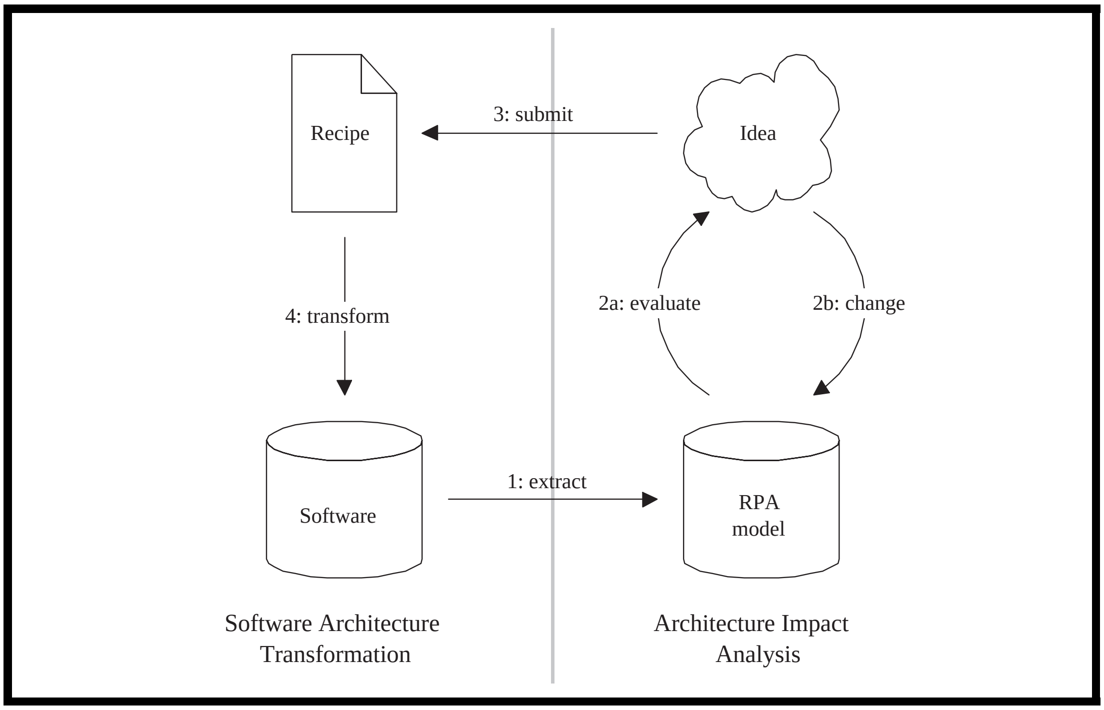
University of Amsterdam
Programming Research Group



A Two-phase Process for Software Architecture Improvement

René Krikhaar
André Postma
Alex Sellink
Marc Stroucken
Chris Verhoef

Report P9903 april 1999

University of Amsterdam
Department of Computer Science
Programming Research Group

A Two-phase Process for Software Architecture Improvement

René Krikhaar
André Postma
Alex Sellink
Marc Stroucken
Chris Verhoef

Report P9903 april 1999

René Krikhaar

Philips Research Laboratories
Prof. Holstlaan 4 (WL01)
5656 AA Eindhoven
The Netherlands
e-mail: krikhaar@natlab.research.philips.com

André Postma

Philips Research Laboratories
Prof. Holstlaan 4 (WL01)
5656 AA Eindhoven
The Netherlands
e-mail: postmaa@natlab.research.philips.com

M.P.A. Sellink

Programming Research Group
Department of Computer Science
University of Amsterdam

Kruislaan 403
NL-1098 SJ Amsterdam
The Netherlands

tel. +31 20 525 7583
e-mail: alex@wins.uva.nl

Marc Stroucken

Philips Research Laboratories
Prof. Holstlaan 4 (WL01)
5656 AA Eindhoven
The Netherlands
e-mail: stroukn@natlab.research.philips.com

C. Verhoef

Programming Research Group
Department of Computer Science
University of Amsterdam

Kruislaan 403
NL-1098 SJ Amsterdam
The Netherlands

tel. +31 20 525 7581
e-mail: x@wins.uva.nl

Universiteit van Amsterdam, 1999

A Two-phase Process for Software Architecture Improvement

- Research Paper -

René Krikhaar*, André Postma*, Alex Sellink**, Marc Stroucken*, Chris Verhoef**

*Philips Research Laboratories, Prof. Holstlaan 4 (WL01),
5656 AA Eindhoven, The Netherlands

**University of Amsterdam, Programming Research Group
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

{krikhaar,postmaa,strouckn}@natlab.research.philips.com, {alex|x}@wins.uva.nl

Abstract

Software architecture is important for large systems where it is the main means for, among other things, controlling complexity. Current ideas about software architectures were not available more than ten years ago. Software that was developed at that time, has been deteriorating from an architecture point of view over the years, as a result of adaptations that were made to the software because of changing system requirements. Still parts of the old software are being used in new product-lines. To make changes to that software, like adding features, it becomes imperative to first adapt the software to accommodate those changes. Software architecture improvement on the existing software is therefore becoming more and more important.

This paper describes a two-phase process for software architecture improvement, which is the synthesis of two research areas: the architecture visualisation and analysis area of Philips Research, and the transformation engines and renovation factories area of the University of Amsterdam. Software architecture transformation plays an important role, and is a new research topic as far as we know. Phase one of the process is based on Relation Partition Algebra (RPA). By lifting the information to a higher level of abstraction and calculating metrics over the system, all kinds of quality aspects can be investigated. Phase two is based on formal transformation techniques on abstract syntax trees. The software architecture improvement process allows for a fast feedback loop on results, without having to deal with the complete software and without interfering with the normal development process.

Keywords: software architecture, software recovery, software rearchitecting, software architecture transformation

1 Introduction

Royal Philips Electronics N.V. is a world-wide company that develops low-volume professional systems (such as communication systems and medical systems) and high-volume consumer electronics systems (like digital set-top boxes and television sets). Software plays an increasing role in all these systems.

In the domain of high-volume electronics the turning point has been reached, where it is no longer possible to develop each new product from scratch. The software architecture of such a new product plays an increasing role to satisfy the demands for increasing functionality with decreasing time-to-market. Design for reuse and open architectures are of the utmost importance to enable software architectures for product lines [BCK98].

In the domain of professional systems this turning point was already reached more than ten years ago. At the time when these large software systems were developed, most of the current software architecture techniques were not available. Still the old software will be used in the development of new products. Current products become more and more feature-driven, which puts a high demand on the flexibility and maintainability of the software.

Since we want to accommodate future changes to the software in both domains, there is a need for software architecture improvement. This paper describes a new software architecture improvement process that combines two research areas.

The main topic of this paper is the description of a two-phase process for recovering and improving software architectures, with a clear separation between the architecture impact analysis (phase one) and the software architecture transformations (phase two). Architecture impact analysis uses a model based on Relation Partition Algebra (RPA).

Software architecture transformations use formal transformation techniques, and aim at modifying the software such that it meets the new software architecture requirements.

A model based on RPA is used to perform impact analysis. Formal transformation techniques are used to modify the software to meet the new architecture requirements.

The paper is structured as follows. In Section 2 a global description is given of the process for software architecture improvement. Section 3 describes an example to illustrate this process. In Section 4 issues related to architecture impact analysis are discussed. Section 5 focuses on the software architecture transformations and describes ideas for a set of basic transformations that are of interest for these kinds of software architecture improvements. Finally, related work is described and some conclusions are given.

2 Improvement Process

We need to address a number of questions for the type of software systems under investigation:

- How do we make software architectures of existing systems explicit?
- How do we exercise and measure improvement?
- How can we perform changes to the software, without introducing new defects and without spending too much time?

The process that we propose that can answer these questions is depicted in Figure 1. Before we give a detailed description of the steps, we give some definitions and assumptions on which this process was built. We adhere to terminology as defined by Chikofski and Cross [CC90].

To improve software architecture we must first have a *described software architecture*, which consists of an explicit description of requirements of the system from a software architecture point of view [SNH95, Kru95]. For large software systems a software architecture description is usually not available. *Software architecture recovery* or *reverse architecting* [Kri97] is the process that extracts such a description from the software.

Rearchitecting is the process of changing the software architecture. *Software architecture improvement* is the process that makes changes to the architecture in such a way that it improves the software in one or more of its quality aspects. *Quality aspects* are for instance the comprehensibility of the software for the developers, the extensibility of the software with new features and the reusability of its parts. Quality aspects are usually accompanied by *metrics*. Using the proper metrics we can *measure* improvement by comparing the values before and after the change. Defining good metrics is one of the research topics, which is beyond the scope of this paper. To experiment with different kinds of metrics and changes we need a good process, supported by tooling.

Changes can affect many parts of the software. Before a change is executed, an architect must know exactly which parts of the software will be affected. Also the cost of implementing the change is important. *Architecture impact analysis* is the process of calculating the consequences of an architecture change before applying it to the software. If the architect can track the impact of a change, then it is also possible to automate the actual changing of the software. Automation can help to increase the quality and reduce the cost of implementing the change. Since we do not want to keep modifying the software during each experiment, it is better to use an abstract model of the software.

At Philips we have many years of experience using Relation Partition Algebra [FKO98, FK99, FO99], which uses a mathematical model based on sets and relations and which has proven to be quite adequate for these kinds of impact analyses. Using a model of the existing software allows for fast feedback of the impact without modifying the software.

The software architecture improvement process that we propose makes a separation between architecture impact analysis on one-hand and software architecture transformations on the other. Figure 1 shows the software architecture improvement process, after which each of the subsequent steps in the process is described. An example using the process is described in Section 3.

Step 1: extract

The software architecture description is *extracted* from the *software* (source code, implementation environment, naming and coding conventions, etcetera). We first have to define what such a software architecture description is. In general, it describes the relations between so-called *design entities*, for instance a use-relation. Design entities are levels of abstraction that constitute some form of hierarchy or part-of relation, like layers, components, modules, functions, etcetera. Each software architecture has its own terminology. The result of the *extract*-step is an *RPA model*.

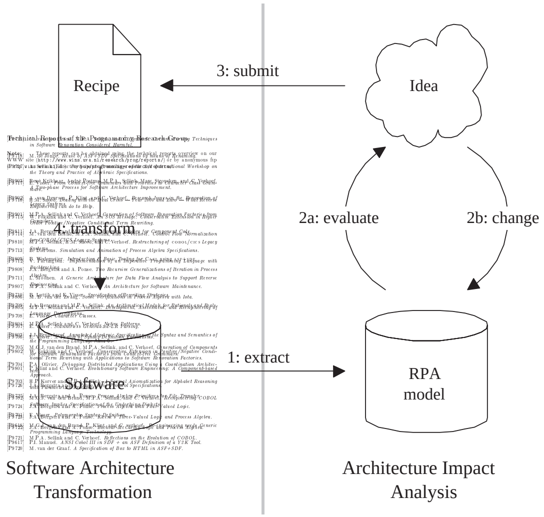


Figure 1: Software Architecture Improvement

Step 2a: evaluate

The *RPA model* is *evaluated*. The *evaluation* gives an image of the software architecture as it is reflected by the *RPA model* of the *software*. Metrics corresponding to quality aspects can be calculated. Also, structures can be visualised, for instance using box-arrow diagrams. The architect can now form *ideas* on how to *change* the *RPA model* in order to improve it. Although actions like calculating metrics and building structure diagrams can be automated, the actual decisions must be made in an interactive process with the architect.

Step 2b: change

Changes are made to the *RPA model*. Making changes to an abstract model of the *software* (like RPA) makes it easier to try *ideas*, contrary to making the changes to the *software* directly. The results are faster available and the *software* does not get corrupted in the process. The architect can try different changes and use backtracking if a change does not lead to the desired improvement of the software architecture. A subsequent *evaluate*-step results in metrics or diagrams of the new model that can be compared to the old model. Several changes can be stacked by repetitively executing a *change* followed by an *evaluate*-step.

Step 3: submit

The *changes* that have been performed on the *RPA model* are now used to *submit* a *recipe*. A *recipe* is an ordered list of *basic transformations* that have to be performed on the *software*. Given a set of basic transformations for each possible change to the *RPA model*, the *submit*-step is rather trivial. The architect can decide to *submit* several of these recipes, before *transforming* the *software*. However, eventually the *recipes* must be executed in order of creation.

Step 4: transform

A *recipe* is used to actually *transform* the *software*. The basic transformations in the *recipe* are executed in order of occurrence. In the implementation, each of these basic transformations can be subdivided into several language dependent transformations. Automating the *transform*-step of the software architecture improvement process eliminates errors otherwise caused by humans and is much faster.

Once the *transform*-step is completed for all *recipes* the *software* again reflects the *RPA model*, as if the *RPA model* was *extracted* from the new *software*. The process for software architecture improvement can start again, but the *extract*-step can be skipped, assuming an error-free tool chain. Note that by logging the transformations made to the *software* the architect can present the changes to the software crew that is reponsible for the maintenance.

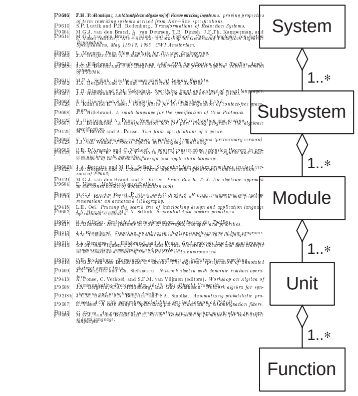


Figure 2: Example: Hierarchical Model

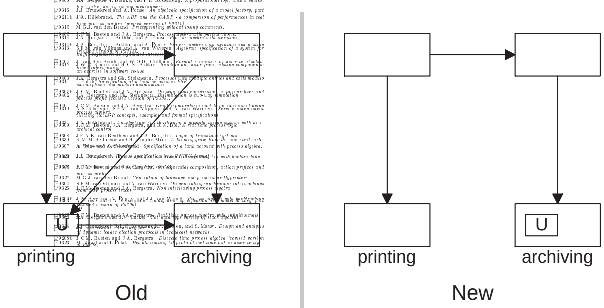


Figure 3: Module Use-Relation

3 Example

We want to clarify the process described in Section 2, with an example that reflects aspects of a real system. Let us consider a system that is hierarchically decomposed into the following design entities: subsystems, modules and units. In Figure 2 the decomposition is described in a UML [Fow97] class model, which states that a subsystem consists of one or more modules.

The system consists of for instance 1518 units containing software, which are written in different programming languages, to name a few: Fortran, Pascal, PL/M, Chill, SDL, SQL, Prom, several assemblers, Make, RCS and Script. Units use each other by means of for instance calling a function. An import statement reflects a use-relation between units, for instance in the programming language C [KR88] `unit7.c` can contain a statement

`#include "unit3.h"`.

This means that *unit7* uses *unit3*. (In C a unit consists of a header file, containing function declarations, and a body file, containing function definitions.) The system decomposition is expressed in a part-of relation, for instance $<$*unit7*,*mod3*$>$ is a member of that relation, which means that *unit7* is contained in *mod3*.

Idea 1:

A unit U, containing archiving related functionality, is located in the module *printing* and the architect considers moving unit U to the module *archiving*. To simulate the consequences of this change we modify the RPA model by updating the part-of relation: $<$$U$,*printing*$>$ is replaced by $<$$U$,*archiving*$>$. In Figure 3 we see on the left hand side the original use-relation on module level and on the right hand side we see the new (simulated) view of the system.

After the evaluation of the result, the architect decides to transform his or her idea to a recipe for changing the software. To explain the required modification in the software we first have to discuss how the design entities map onto the elements of the software archive, including a discussion of the build process.

Figure 4 shows a UML picture of the relationships of the build description. A version of the software consists of

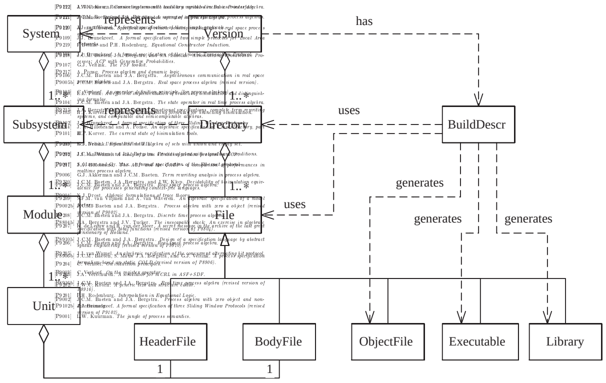


Figure 4: Build Description

a number of directories in which files reside. Furthermore, a build description belongs to a version, which describes how objects, executables and libraries are created from the software files.

For the programming language C a unit consists of a header file and a body file. A module is not explicitly available in the archive, however a filename prefix refers to the module name. A subsystem is reflected by a directory and a system maps onto a version. To move unit U from module *printing* to the module *archiving*, we have the following recipe of changes.

Recipe 1:

1. Rename the files of unit U so that they get the proper prefix.
2. Move the files of unit U to the directory (subsystem) to which the module *archiving* belongs.
3. Change all import statements in all the units according to the newly created name of the header file of unit U.
4. Adapt the build description (change file names, adapt include paths, etcetera).

Following the recipe someone can modify the software by hand, but this is a tedious and error prone task. For large systems, the build description may consist of dozens of makefiles, which all must be analysed and possibly adapted. Also, in our case 1518 units may need to be adapted for the proper import statements. Already for this relative simple architectural transformation we need tools to change the software.

Idea 2:

In legacy systems, functions may be organised in an arbitrary unit. During construction, a developer is sometimes pressured by time constraints to put a specific function in one unit while semantically it would fit better in another unit. The architect has decided to move the function to the correct unit, say function F in unit U to unit X. The detailed recipe for applying this idea to the software heavily depends on for instance the implementation languages. Let us again take the programming language C as example, then more specific the recipe contains the following entries.

Recipe 2:

1. Move the declaration and definition of function F to the header file and body file of unit X respectively.
2. Change the `#include` statements in files that use function F.
3. Adapt the build description files.

Ad.1 Moving the function declaration and definition is not trivial, also the types used in the function have to be within the scope of the new location, which means relocating type definitions or adding additional `#include` statements. The same goes for global variables and macros that are being used.

Idea 3:

Several units in the software contain identical functions or so-called *function-clones*. The architect considers to remove all but one, and change the use-relation of the units accordingly. The architect uses metrics for cohesian and coupling to determine which of the function-clones should remain in the software. Once he has determined which function is the best (by trying one and using backtracking before trying the next), the following recipe can be executed.

Recipe 3:

1. Change the `#include` statements in the files that use one of the functions that will be removed.
2. Remove the function-clones.
3. Adapt the build description.

The impact analysis of this kind of change has been exercised and implemented in the Abstract-level Re-clustering Tool [Bro99] developed at Philips Research, which calculates the impact of clone elimination and re-clustering on the cohesian and coupling metrics.

For embedded systems we should also consider the target system files (executables, dynamic libraries, scripts). These files are generated during construction and copied to the appropriate location on the target (for instance an EPROM). In a first experiment we decided to keep the target executables the same, i.e. not to change the communication protocols between the executables (which may occur if the architect moves a function from one target to another). In a next step, it is possible to also model the execution view of the systems, including the communication, after which the impact of these kind of changes can also be viewed.

In this example we gave three ideas that were of interest to an architect after inspecting the views on the model (like Figure 3). One can also consider more automation by introducing architectural metrics. Without knowledge about the semantics of the system these metrics indicate some quality aspect of the system.

4 Architecture Impact Analysis

Software architecture impact analysis is concerned with measuring change on an architectural level. In our process, this not only means visualising the parts of the architecture that are affected by a change, but also comparing the before and after situations. Impact analysis covers steps 2a and 2b of Figure 1. In these steps the software architect interacts with the RPA model. If he or she sees possibilities for improvements, changes can be made to the model and the results can be evaluated and compared with the original. Before the software architect can start interacting with the RPA model, information has to be extracted from the software. The extraction is mostly an automated process.

For extracting information from the software, scripts can be used, often in combination with commercial tools, like QAC [Pro96], Sniff [Tak95] and the Microsoft BSC-kit. The type of information that is extracted, is for instance the use-relation between functions, otherwise known as the call-graph. Also the part-of relation for each level of abstraction (Figure 2) can be extracted. Following the example, the part-of relation between units and modules can be extracted by scanning the filenames of the units for their prefix. Simpler is the part-of relation between modules and subsystems, which relates to files in directories. Note that the same module prefix may have been used in different directories, which in the example is considered an architecture violation that needs to be corrected in a first transformation.

Using RPA expressions, it is possible to add high-level information to the RPA model. For instance, the use-relation between units can be calculated by lifting the use-relation between functions, using the part-of relation between functions and units. RPA has a rich set of operators to calculate the high-level information [FKO98]. Also, adding accounting information is possible, so that the architect knows how many static function calls exist from one unit to another [FK99]. Figure 5 shows an example. The left hand side shows a use-relation of units. The right hand side shows the lifted use-relation (module-level) with accounting information.

One form of evaluating the recovered architecture, is using metrics for the different quality aspects. Metrics are defined using an RPA expression, which can be executed on the RPA model. An example of a metric is the cohesion of a unit, which can be expressed as the quotient of the number internal static function calls and the number of possible internal static function calls. Some other examples of metrics are listed below.

- A metric for coupling defines the rate of external connections. The coupling-metric, like the one for cohesion, can be defined for different levels of abstraction.

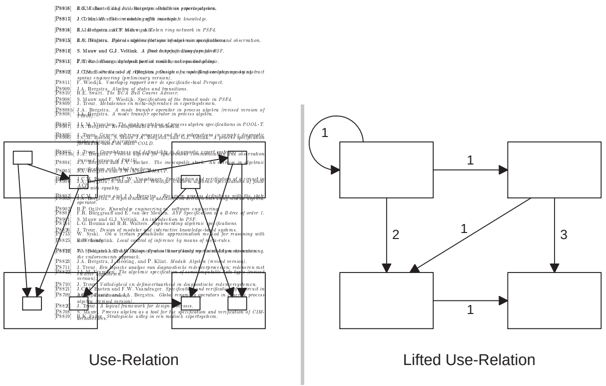


Figure 5: (Multi) Use-Relation

- A metric for layering indicates the rate of up-calls in a layered architectural model.

Another form of evaluating the recovered architecture is using box-arrow diagrams like in Rigi [SWM97], the Software Bookshelf [FHK+97b] and 3D visualisation [FJ98]. In Figure 3 we already saw how the old and new situation of a part of the use-relation between modules are compared. Restricted views and zoom functions can be calculated to let the software architect focus on the problems at hand. Note that architecture impact analysis is a highly interactive process that can not be automated. The final decisions have to be made by the architect. The process can only propose changes. An example of a valuable change to the model is function-clone elimination as described in the example of Section 3.

The process for software architecture improvement, supported by the proper tooling, gives us an appropriate framework for investigating the impact of changes on an architectural level, as well as a basis for defining new and better metrics for the different quality aspects.

5 Software Architecture Transformations

The goal of impact analysis performed on a model of the software architecture is to identify valuable architecture changes, i.e. changes that lead to an improved software architecture. Once one or more valuable architecture changes have been identified on the model, we want to adapt the software of the existing system accordingly. For this purpose a recipe is used, which describes how to *translate* changes performed on the model of the software architecture into changes to the actual software.

A recipe is a generic description of a so-called architecture transformation. An *architecture transformation* is a transformation to the software, which has an impact on the architectural model of the system.

An interesting property of architecture transformations is that they are composable, so larger architecture transformations can be constructed as a sequence of smaller architecture transformations (see the example below).

Our goal is to identify a set of useful architecture transformations. This set will include *basic architecture transformations* (i.e. architecture transformations that are not described as a sequence of smaller architecture transformations) and *composite architecture transformations.* Composite transformations consist of a number of basic transformations, but have a right of existence of their own (mostly because they are frequently used). We expect this set of useful architecture transformations to grow in time as we gain more insight in what kind of transformations are needed. The basic and composite architecture transformations can be seen as building blocks, out of which recipes can be constructed.

There are different advantages of constructing recipes as sequences of architecture transformations over constructing them by hand.

- An architecture transformation provides a *standard* solution, which can be reused. Constructing a recipe for a certain change, as a fixed sequence of architecture transformations, guarantees that this change is always performed in the same way. This probably makes it easier to understand the architecture of the transformed software.

- Architecture transformations are building blocks, hence, recipes can be composed as a sequence of architecture transformations, which saves development time.

- Implementing small basic architecture transformations is much simpler than implementing complete recipes.

We will now give some examples of possible architecture transformations, both basic and composite, based on the example given in Section 3.

`CreateUnit`-transformation
This basic transformation creates the files for a new and empty unit U. Since the unit is not used anywhere yet, creating it does not have an impact on the build description. For the programming language C, a new body file `U.c` and header file `U.h` are created.

`DeleteUnit`-transformation
This basic transformation deletes the files of a unit U. This is a dangerous transformation if the functions in unit U are still being used. If this is not the case, the build description will not be effected. For the programming language C, the body file `U.c` and the header file `U.h` are deleted.

`RenameUnit`-transformation
This basic transformation renames the files of a unit U to a unit V. This is a dangerous transformation. If there is already a unit called V, it will be overwritten. Every unit X that uses functions of unit U needs to import unit V instead. For the programming language C, the `#include "U.h"` is replaced by `#include "V.h"` in the files of unit X. In the build description, every occurrence of unit U is replaced by V.

`IsolateFunction`-transformation
This basic transformation isolates one function F from unit U and relocates it to an existing empty unit V. Every unit X that uses function F, needs to import unit V. If unit X does not use any other functions of unit U, the import of unit U can be removed. For the programming language C, the statement `#include "V.h"` is added to the files of unit X. In the build description, a dependency with unit V is added to unit X.

`CombineUnits`-transformation
This basic transformation combines two units U and V to an existing empty unit W. This is a dangerous transformation, if W is one of U or V. Every unit X that uses functions from either unit U or V, needs to replace the imports with an import of unit W. For the programming language C, the statements `#include "U.h"` and `#include "V.h"` are removed from the files of unit X, and a statement `#include "W.h"` is added. In the build description, every occurrence of unit U and unit V is replaced by W.

`MoveFunction`-transformation
This composite transformation moves a function F from unit U to unit V. It can be constructed using the previously defined basic transformations.

1. `CreateUnit` T_1
2. `IsolateFunction` F from unit U in unit T_1
3. `CreateUnit` T_2
4. `CombineUnits` T_1 and V into unit T_2
5. `DeleteUnit` T_1
6. `DeleteUnit` V
7. `RenameUnit` T_2 to unit V

Once the ideas are evaluated and turned into recipes, they can be expressed in architecture transformations like we have seen above. So in principle it is now clear what has to be done to the software. To implement the transformations is another issue. In order to make automatic changes to the complete software, we implement a software renovation factory. This is an architecture that enables rapid implementation of tools that are typical for code changes. The construction of such factories is beyond the scope of this paper. The interested reader is referred to [BKV96, KV98, BKV98, DKV99, SV99a, SV99b, BD99].

The systems that we are dealing with are mixed language applications. We have to construct a *reengineering grammar* that understands all applied languages. In [SV99b] a process and tooling are described to generate a reengineering grammar from the source code of compilers. In that paper, 20 sublanguages played a role. When this grammar is defined, we can generate software that we call a software *renovation factory*, see [SV99a] for an overview and [BD99] for applications. The set-up of such a factory is not trivial, and takes some time.

In order to be able to carry out basic architecture transformations, some of the software must be annotated with so-called *scaffolding* information. For example, when a function is moved to another file, we can annotate all calls to this function in the source files so that we know in a later stage that we can turn those calls into other calls. It will be clear that the architecture transformations are implemented as complete *assembly lines* containing many small steps.

To give an example, when we generate a software renovation factory from a modular grammar, the entire system consists of grammar files, and the transformation is called `Gen-Factory`. The implementation parses the entire grammar system using scaffolding and about 200 different steps the entire system wide transformation is carried out. Similar principles apply to the architecture transformations for systems that do not consist of grammar files.

6 Related Work

In Chapter 19 *Software Architecture in the Future* of the textbook [BCK98] architecture migration technology is mentioned. Our paper is a first step towards this migration technology of the future, and as far as we know, this is a new subject. Of course, parts of the subject of artchitectural transformations do exist. We mention a few efforts to analyse software architectures: Rigi [SWM97], The Software Bookshelf [FHK+97a], Dali [KC99], and efforts at Philips Research Labs [Kri97, KPZ99]. On the code level also a lot of work has been done, we mention Sneed's Reengineering Workbench [Sne98], the TAMPR system [BHW96], TXL [CHHP91], REFINE [Rea92], COSMOS [Eme98], The ASF+SDF Meta-Environment [Kli93], Elegant [Aug93, Phi93].

The combination of architecture analysis tools and tools that work on the code level is an issue that gained attention at SEI under the name CORUM [WL+98, KWC98]. This effort aims at the interoperation of tools by a common format. Our approach differs from theirs in the sense that we do not focus on interoperability, like sharing a parser both for architecture impact analysis and for code transformations. Instead of reusing parser output, we focus on reusing the source code of the parser, so that we can change the parser into a parser appropriate for reengineering [SV99b].

Our approach is more in line with the COSMOS approach for solving the Y2K problem. After analysis, COSMOS returns a prescription of what needs to be done to the code. This prescription can be carried out by hand, or in some cases be executed automatically. A difference is, we focus on the transformation's impact on the architecture, which is mostly not the case for Y2K repair engines.

In the impact phase (phase one) we adapt our RPA model in order to evaluate the effect of modifications of the architecture beforehand. This is comparable to the *Software Architecture Analysis Method* or SAAM [BCK98, Ch. 9]. The mathematical foundations of RPA [FO99] are similar to the mathematical foundations of [Hol98].

7 Conclusions

The process for software architecture improvement that we propose in this paper is completely implementation independent. The techniques that we combine still leave room for other possibilities. We have chosen Relation Partition Algebra as an abstract model of the software, because we have had good experiences with the model and its usefulness in many applications. Related work like for instance in [Hol98] can also be used. The architecture transformations are implementation independent themselves, because they are expressed in smaller programming language specific transformations. For the implementation of these language specific transformation any kind of compiler technology can be used. We named a few in Section 6. We have chosen the software factory technology with its assembly lines, because it comes close to our ideas about decomposable transformations.

We have shown that the process for software architecture improvement is flexible in experimenting with new metrics and changes, in order to find those best suited for the systems that we investigate. By forming ideas, trying them on the model we can do an early impact analysis. The changes made to the model are submitted in recipes containing high level architecture transformations, which themselves consist of basic and composite transformations.

The process for software architecture improvement separates architecture impact analysis from architecture transformations, which in contrary to performing changes to the software directly, gives us the following advantages.

- During idea generation there is no interference with daily work of software developers.
- Impact analysis is performed on an abstract view of the system, which heavily reduces amount of information. We can gain insight, without being clouded by the details of the software.
- Making changes to a model gives the opportunity to backtrack in the flow of ideas.
- The feedback of the changes is almost immediate.
- Last but not least, the process is open for future research results on appropriate metrics and changes.

The feasibility of the approach has already been proven in [Bro99] for clone elimination. In our future work we want to excercise and implement this process for software architecture improvement. The work will consist of the following steps.

- Develop an impact analysis system on top of existing RPA functionality.
- Implement a framework for logging and backtracking changes.
- Define proper basic architecture transformations.
- Implement the basic architecture transformations.
- Evaluate the process in a real world system.

References

[Aug93] L. Augusteijn. *Functional Programming, Program Transformations and Compiler Construction.* PhD thesis, Eindhoven University of Technology, 1993.

[BCK98] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice.* Addison-Wesley Publishing Company, 1998.

[BD99] J. Brunekreef and B. Diertens. Towards a user-controlled software renovation factory. In P. Nesi and C. Verhoef, editors, *Proceedings of the Third European Conference on Maintenance and Reengineering*, pages 83–90, 1999.

[BHW96] J.M. Boyle, T.J. Harmer, and V.L. Winter. The TAMPR program transformation system: Design and applications. In *The SciTools'96 Electronic Proceedings*, page 13 p., 1996. http://www.oslo.sintef.no/SciTools96/Contrib/boyle/scitlpap.912.ps.

[BKV96] M.G.J. van den Brand, P. Klint, and C. Verhoef. Core technologies for system renovation. In K.G. Jeffery, J. Král, and M. Bartošek, editors, *SOFSEM'96: Theory and Practice of Informatics*, volume 1175 of *LNCS*, pages 235–255. Springer-Verlag, 1996.

[BKV98] M.G.J. van den Brand, P. Klint, and C. Verhoef. Term rewriting for sale. In C. Kirchner and H. Kirchner, editors, *Second International Workshop on Rewriting Logic and its Applications*, Electronic Notes in Theoretical Computer Science. Springer-Verlag, 1998. Available at: http://adam.wins.uva.nl/~x/sale/sale.html.

[Bro99] J.W. Brook. Design and implementation of a tool for re-clustering. Master's thesis, Eindhoven University of Technology, department of Mathematics and Computer Science, 1999.

[CC90] E. Chikofsky and J. Cross. Reverse Engineering and Design Recovery: A taxanomy. *IEEE Software*, pages 13–17, January 1990.

[CHHP91] J.R. Cordy, C.D. Halpern-Hamu, and E. Promislow. TXL: A rapid prototyping system for programming language dialects. *Computer Languages*, 16(1):97–107, 1991.

[DKV99] A. van Deursen, P. Klint, and C. Verhoef. Research issues in the renovation of legacy systems. In J.-P. Finance, editor, *Fundamental Approaches to Software Engineering*, LNCS. Springer-Verlag, 1999. Available at: http://adam.wins.uva.nl/~x/etaps/etaps99.html.

[Eme98] Emendo Software Group, The Netherlands. *Emendo Y2K White paper*, 1998. Available at http://www.emendo.com/.

[FHK+97a] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Muller, J. Mylopoulos, S. Perelgut, M. Standley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, 1997.

[FHK+97b] P.J. Finnigan, R.C. Holt, I. Kalas, S. Kerr, and K. Kontogiannis. The software Bookshelf. *IBM Systems Journal*, 36(4):564–593, 1997.

[FJ98] L. Feijs and R.P. de Jong. 3D visualization of software architectures. *Communications on the ACM*, 41(12):73–78, December 1998.

[FK99] L.M.G. Feijs and R.L. Krikhaar. Relation Algebra with Multi-Relations. *International Journal Computer Mathematics*, 70:57–74, 1999.

[FKO98] L. Feijs, R. Krikhaar, and R. van Ommering. A relational approach to Software Architecture Analysis. *Software Practice and Experience*, 28(4):371–400, April 1998.

[FO99] L.M.G. Feijs and R.C. van Ommering. Relation Partition Algebra – mathematical aspects of uses and part-of relations –. *Science of Computer Programming*, 33:163–212, 1999.

[Fow97] Martin Fowler. *UML Distilled - applying the standard object modeling language*. Addison-Wesley Publishing Company, 1997.

[Hol98] Richard C. Holt. Structural Manipulations of Software Architecture using Tarski Relational Algebra. In *Proceedings of Fifth Working Conference on Reverse Engineering*. IEEE Computer Society, 1998.

[KC99] R. Kazman and J. Carrière. Playing detective: Reconstructing software architecture from available evidence. *Journal of Automated Software Engineering*, april 1999. To appear.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

[KPZ99] R.L. Krikhaar, M.P. Pennings, and J. Zonneveld. Employing use-cases and domain knowledge for comprehending resource usage. In P. Nesi and C. Verhoef, editors, *Proceedings of the Third European Conference on Maintenance and Reengineering*, pages 14–21, 1999.

[KR88] B. Kernighan and D.M. Ritchie. *The C programming Language*. Prentice Hall, second edition, 1988.

[Kri97] R.L. Krikhaar. Reverse Architecting Approach for Complex Systems. In *Proceedings International Conference on Software Maintenance*, pages 4–11. IEEE Computer Society, 1997.

[Kru95] P. Kruchten. The $4+1$ View Model of Architecture. *IEEE Software*, pages 42–50, November 1995.

[KV98] P. Klint and C. Verhoef. Evolutionary software engineering: A component-based approach. In R.N. Horspool, editor, *IFIP WG 2.4 Working Conference: Systems Implementation 2000: Languages, Methods and Tools*, pages 1–18. Chapman & Hall, 1998. Available at: http://adam.wins.uva.nl/~x/evol-se/evol-se.html.

[KWC98] R. Kazman, S. Woods, and J. Carrière. Requirements for integrating software architecture and reengineering models: Corum ii. In M.H. Blaha, A. Quilici, and C. Verhoef, editors, *Proceedings Fifth Working Conference on Reverse Engineering*, pages 154–163, 1998.

[Phi93] Philips Electronics B.V., The Netherlands. *The Elegant Home Page*, 1993. http://www.research.philips.com/generalinfo/special/elegant/elegant.html.

[Pro96] Programming Research Ltd. *QAC Version 3.1 User's Guide*, 1996.

[Rea92] Reasoning Systems, Palo Alto, California. *Refine User's Guide*, 1992.

[Sne98] H.M. Sneed. Architecture and functions of a commercial software reengineering workbench. In P. Nesi and F. Lehner, editors, *Proceedings of the Second Euromicro Conference on Maintenance and Reengineering*, pages 2–10, 1998.

[SNH95] D. Soni, R. Nord, and C. Hofmeister. Software Architecture in Industrial Application. In *Proceedings International Conference on Software Engineering*, pages 196–207, 1995.

[SV99a] M.P.A. Sellink and C. Verhoef. An architecture for automated software maintenance. In *Proceedings of the seventh International Workshop on Program Comprehension*, 1999. Available at http://adam.wins.uva.nl/~x/asm/asm.html.

[SV99b] M.P.A. Sellink and C. Verhoef. Generation of software renovation factories from compilers. Technical Report P9901, University of Amsterdam, Programming Research Group, 1999. Available via http://adam.wins.uva.nl/~x/com/com.html.

[SWM97] Margaret-Anne D. Storey, Kenny Wong, and Hausi A. Mueller. Rigi: A Visualisation Environment for Reverse Engineering. In *Proceedings of International Conference on Software Engineering*, pages 606–607, 1997.

[Tak95] TakeFive Software. *SNiFF+ – User's Guide and Reference*, 1995.

[WL+98] S.G. Woods, L. O'Brian T. Lin, , K. Gallagher, and A. Quilici. An architecture for interoperable program understanding tools. In *Proceedings of the sixth International Workshop on Program Comprehension*, pages 54–63, 1998.

[P8707] J.H. Verhagen. *Expertsystemen bij de Nederlandse Spoorwegen.*

[P8706] J.C.M. Baeten, J.A. Bergstra, and J.L.M. Vrancken. *Processen en procesexpressies.*

[P8705] J.L.M. Vrancken. *The algebraic specification of semicomputable datatypes.*

[P8704] F. Wiedijk. *Termherschrijfsystemen in Prolog.*

[P8703] A.V. Hurkmans. *Een declaratieve en procedurele kennisrepresentatievorm voor kennissystemen, toegepast op NEXT.*

[P8702] M.R. Dasselaar. *Development of an expert system, from theory to practice.*

[P8701] R.A. Groenveld. *Verification of a sliding window protocol by means of process algebra.*