A Two-phase Process for Software Architecture Improvement

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 in which it is the main means for, among other things, controlling complexity. Current ideas on software architectures were not available more than ten years ago. Software developed at that time has been deteriorating from an architectural point of view over the years, as a result of adaptations made in the software because of changing system requirements. Parts of the old software are nevertheless still being used in new product lines. To make changes in that software, like adding features, it is imperative to first adapt the software to accommodate those changes. Architecture improvement of 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 to our knowledge a new research topic. Phase one of the process is based on Relation Partition Algebra (RPA). By lifting the information to higher levels 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 the need to deal with the complete software and without any interference 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 increasingly important role in all these systems.

In the domain of high-volume electronics the point has been reached at which it is no longer possible to develop each new product from scratch. The software architecture of new products is of great importance with respect to satisfying the demands for increasing functionality with decreasing time-to-market. Design for reuse and open architectures are of the utmost importance with respect to 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. The old software is nevertheless still used in the development of new products. Current products are more and more feature-driven, and must therefore meet high requirements with respect to the flexibility and maintainability of the software.

Since we want to accommodate future changes in 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 distinction between architecture impact analysis (phase one) and software architecture transformations (phase two). Architecture impact analysis uses a model based on Relation Partition Algebra (RPA [FO94, FKO98, FO99, FK99]). Software architecture transformations use formal transformation techniques, and aim at modifying the software to meet the

new architecture requirements [BKV96, BSV97, SV98, BKV98, DKV99, SV99a, SV99b, SV99c, SV99d].

The paper is structured as follows. In Section 2 a general description of the process for software architecture improvement is given. 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.

Acknowledgements

The authors would like to thank Reinder Bril, Loe Feijs, Peter van den Hamer, Jaap van der Heijden and Joachim Trescher for reviewing previous versions, and the fruitful discussions that helped us to improve this paper.

2 Improvement Process

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

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

The process which in our opinion can answer these questions is depicted in Figure 1. Before we give a detailed description of the steps, we will give some definitions and assumptions on which this process is based. We adhere to the terminology proposed by Chikofsky and Cross [CC90].

To improve software architecture we must first have a *described software architecture*, which is 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, Kri99] 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 in 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 a research topic 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 important is the cost of implementing the change. *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 [FO94, FKO98, FO99, FK99], which involves a mathematical model based on sets and relations and has proven to be quite adequate for impact analyses of this kind. Using a model of the existing software allows for fast feedback of the impact without the need to modify the software.

The software architecture improvement process that we propose makes a distinction between architecture impact analysis on the one hand and software architecture transformations on the other. Figure 1 shows the software architecture improvement process. Each of the steps in the process (corresponding to the numbers in the figure) will be described successively. An example using the process will be 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) and from the architects by conducting interviews. 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. Each software architecture has its own terminology. The result of the *extract* step is an *RPA model*.

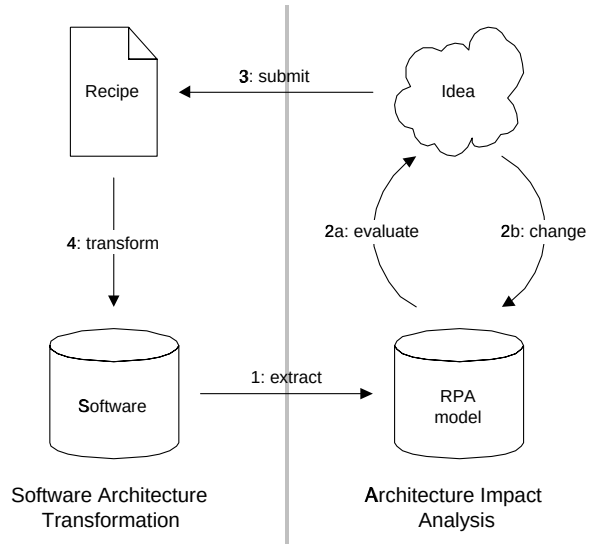

Figure 1: Software Architecture Improvement

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 one or more of the quality aspects. 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 in the *RPA model*. Making changes in an abstract model of the *software* (like RPA) makes it easier to try out *ideas*, rather than make the changes in the *software* directly. The results are available more quickly and the *software* is not 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 with 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 made in the *RPA model* are now used to *submit* a *recipe*. A *recipe* is an ordered list of *transformations* that have to be performed on the *software*. Once we have developed a set of basic transformations for each useful change in the *RPA model*, the *submit* step is rather trivial. Finding good basic transformations will be a major topic of future research. The architect can decide to *submit* several of these recipes, before starting the *transform* step. However, eventually the *recipes* must be executed in the order of their creation.

Step 4: transform

A *recipe* is used to *transform* the *software*. The basic transformations in the *recipe* are executed in the order of their 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 has been completed for all *recipes*, the *software* again reflects the *RPA model* in such a way as if the *RPA model* had been *extracted* from the changed *software*. The process for software architecture improvement can start again, but the *extract* step can now be skipped, on the assumption that the tool-chain is error-free. Note that by logging the transformations made in the *software*, the architect can present the changes to the software crew responsible for maintenance.

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 diagram, which states that a subsystem consists of one or more modules. We will use *italics* for the names in the example and `typewriter` style for filenames and code fragments.

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 for instance calling a function. An import statement reflects a use relation between

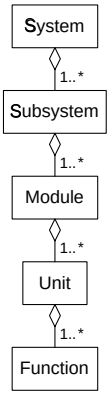

Figure 2: Example: Hierarchical Model

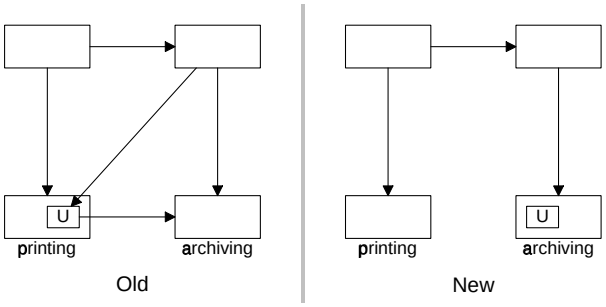

Figure 3: Module Use-Relation

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 the pair <*unit7*, *mod3*> is a member of that relation, which means that *unit7* is contained in (part of) *mod3*.

Idea 1:

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

After the evaluation of the result, the architect decides to transform his or her idea into a recipe for changing

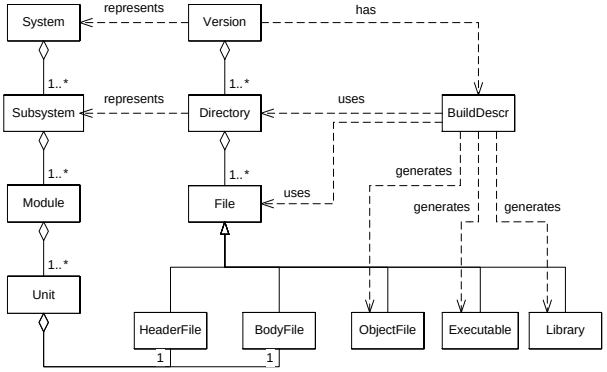

Figure 4: Build Description

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, and explain the build process.

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

We recall that in the programming language C a unit consists of a header file and a body file. A module is not explicitly available in the archive, but 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 the *printing* module to the *archiving* module, we have the following recipe with 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 *archiving* module 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).

It is possible to modify the software by hand by following the recipe, but this is a tedious and error-prone task. For large systems, the build description may consist of dozens of makefiles, which must all be analysed and possibly adapted. Moreover, worst case 1518 units may need

to be adapted to include the proper import statements. For this relatively simple architecture transformation we already 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 decides 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 depends heavily on for instance the implementation languages. Let us again take the programming language C as an example, then more specifically 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.

Referring to step 1 in the recipe: moving the function declaration and definition is not trivial, and 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 holds for global variables and macros that are being used. Although idea 2 is similar to idea 1 (the same change but at a different level of abstraction) the recipes are different. Therefore we must consider these changes to be different. In general we can say that changes are dependent on the applied level of abstraction.

Idea 3:

Several units in the software contain identical functions or so-called *function clones*. The architect considers removing all but one, and changing the use relation of the units accordingly. The architect uses metrics for cohesion and coupling to determine which of the function clones should remain in the software. Once he or she has determined which function is the best choice (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 a change of this kind has been exercised and implemented in the Abstract-level Re-clustering Tool (ART [Bro99]) developed at Philips Research, which calculates the impact of clone elimination and re-clustering based on the cohesion and coupling metrics.

In the case of 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 experiment, it is possible to also model the execution view of the systems, including the communication, after which the impact of such changes can also be viewed.

In this example we have given three ideas that are of interest to an architect after inspecting the views of the model (like the one presented in Figure 3). One can also consider more automation by introducing algorithms that try several changes using architectural metrics. Architectural metrics also abstract from the details of the system and can indicate some quality aspect of the system at some level of abstraction.

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 in 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.

Scripts can be used to extract information from the software often in combination with commercial tools, like QAC [Pro96], Sniff [Tak95] and the Microsoft BSC kit. The type of information extracted is for instance the use relation between functions, otherwise known as the call-graph. The part-of relation for each level of abstraction (Figure 2) can also be extracted. Following the example,

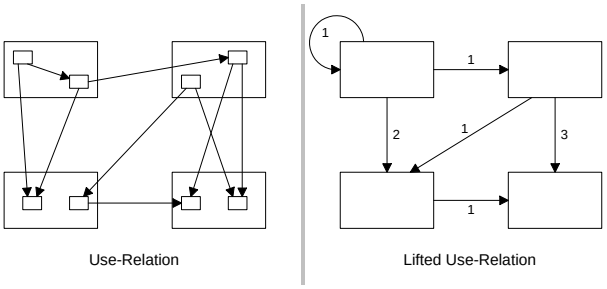

Figure 5: (Multi) Use-Relation

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 [FO94, FKO98, FO99, FK99].

It is also possible to add accounting information by using multi-relations [FK99], so that the architect knows how many static function calls exist from one unit to another. 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 way of evaluating the recovered architecture is using architectural 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 of internal static function calls and the number of all 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 metric for cohesion, can be defined for different levels of abstraction.
- A metric for layering indicates the rate of up-calls in a layered architectural model.

Another way of evaluating the recovered architecture is by using box-arrow diagrams as in Rigi [SWM97], the Software Bookshelf [FHK+97] and 3D visualisation [FJ98]. Figure 3 shows how the old and new situations of a simple use relation between modules are related. Restricting views and zoom functions can be calculated on the RPA model to let the software architect focus on the problems at hand. Note that architecture impact analysis is a highly interactive process that cannot be completely automated. The final decisions have to be made by the architect. The process can only show possible changes and their impact. An example of a valuable change in 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 in 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 in the model of the software architecture into changes in the actual software.

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

An interesting property of architecture transformations is that they can be combined, 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 into the kind of transformations needed. The basic and composite architecture transformations can be seen as building blocks, from which recipes can be constructed.

There are several 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 will always be performed in the same way. This will probably makes it easier to understand the architecture of the transformed software.
- Architecture transformations are building blocks, so 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 dedicated 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 yet used anywhere, creating it does not have an impact on the build description. In 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 given the precondition that the functions in unit U are no longer being used. The build description need not be altered. 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 given the precondition that unit V does not yet exist. If unit V does exist, it will be overwritten. Every unit X that uses functions of unit U needs to import unit V instead. In 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 unit 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. In 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 into an existing empty unit W (not equal to U and V). Every unit X that uses functions from either unit U or V needs to replace the imports with an import of unit W. In 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 unit 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 have been evaluated and turned into recipes, they can be expressed in architecture transformations as we have seen above. So in principle it is now clear what has to be done in the software. Implementing the transformations is another issue. In order to make automatic changes in the complete software, we implement a software renovation factory. This is an architecture that enables rapid implementation of tools that are typical of code changes. The construction of such factories is beyond the scope of this paper. The interested reader is referred to [BKV96, KV98, BKV98, DKV99, SV99b, SV99a, BD99].

The systems we are dealing with are mixed-language applications. We have to construct a *reengineering grammar* that understands all applied languages. [SV99a] describes a process and tools generating a reengineering grammar from the source code of compilers. In that paper, 20 sublanguages play a role. When the grammar has been defined, we can generate software that we call a software *renovation factory*; see [SV99b] for an overview and [BD99]

for applications. Setting up such a factory takes time for systems containing dozens of different languages. The individual grammars of the compilers and scripting facilities need to be reverse engineered. The actual generation of the software factories is automatic.

To be able to carry out basic architecture transformations we can annotate some of the software with so-called *scaffolding* information ([SV99c]). 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 of architecture transformations using scaffolding, 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 the transformation is effected throughout the entire system in about 200 different steps. Similar principles apply to the architecture transformations for systems other than those consisting 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 to our knowledge this is a new subject. Of course, parts of the subject of architecture transformations already exist. To mention a few efforts to analyse software architectures: Rigi [SWM97], The Software Bookshelf [FHK+97], Dali [KC99], and efforts at Philips Research Labs [Kri97, KPZ99]. A lot of work has been done on code level; we mention Sneed's Reengineering Workbench [Sne98], the TAMPR system [BHW96], TXL [CHHP91], REFINE [Rea92], COSMOS [Eme98], RainCode [Rai98], 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 of CORUM [WOL+98, KWC98]. This effort aims at the interoperation of tools by a common format. Our approach differs from theirs in 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 [SV99a].

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 it can be executed automatically. A difference is that we focus on the transformation's impact on the architecture, which is mostly not the case with 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 [FO94, FKO98, FO99, FK99] are similar to the mathematical foundations of [Hol98].

7 Conclusions

The process for software architecture improvement that we have proposed 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 as in [Hol98] can also be used. The architecture transformations are implementation-independent themselves. Only at the lowest level they are expressed in small programming-language-specific transformations. Any kind of compiler technology can be used for the implementation of these programming-language-specific transformations. We named a few in Section 6. We have chosen the software factory technology with its assembly lines because it comes close to our views on 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 we investigate. By forming ideas and trying them out in the model we can perform an early impact analysis. The changes made in the model are submitted as 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 presents the following advantages from making changes in the software directly.

- During idea generation there is no interference with the daily work of the software developers.

- Impact analysis is performed on an abstract view of the system, which greatly reduces the amount of information. We can gain the insight we need, without being distracted by the details of the software.
- Making changes in a model gives us 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 to future research results relating to finding appropriate architectural metrics and changes.

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

- Development of an impact analysis system on top of existing RPA functionality.
- Implementation of a framework for logging and backtracking changes.
- Definition of proper basic architecture transformations.
- Implementation the basic architecture transformations.
- Evaluation of the process for a real-world system.

References

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

[BCK98] L. Bass, P. Clements, and R. 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*, 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. Available at: http://adam.wins.uva.nl/~x/sofsem/sofsem.html.

[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.

[BSV97] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Generation of Components for Software Renovation Factories from Context-free Grammars. In I.D. Baxter, A. Quilici, and C. Verhoef, editors, *Proceedings of the Fourth Working Conference on Reverse Engineering*, pages 144–153, 1997. Available at http://adam.wins.uva.nl/~x/trans/trans.html.

[CC90] E. Chikofsky and J. Cross. Reverse Engineering and Design Recovery: A taxonomy. *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+97] 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.

[FJ98] L.M.G. 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.M.G. Feijs, R.L. Krikhaar, and R.C. van Ommering. A relational approach to Software Architecture Analysis. *Software Practice & Experience*, 28(4):371–400, April 1998.

[FO94] L.M.G. Feijs and R.C. van Ommering. Theory of Relations and its Applications to Software Structuring. Philips internal report, Philips Research, 1994.

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

[Hol98] R.C. Holt. Structural Manipulations of Software Architecture using Tarski Relational Algebra. In A. Quilici and C. Verhoef, editors, *Proceedings of Fifth Working Conference on Reverse Engineering*, pages 210–219. IEEE Computer Society, 1998.

[KC99] R. Kazman and J. Carrière. Playing Detective: Reconstructing Software Architecture from Available Evidence. *Journal of Automated Software Engineering*, 6(2):107–138, April 1999.

[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 M.J. Harrold and G. Visaggio, editors, *Proceedings of the International Conference on Software Maintenance*, pages 4–11. IEEE Computer Society, 1997.

[Kri99] R.L. Krikhaar. *Software Architecture Reconstruction*. PhD thesis, University of Amsterdam, 1999.

[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.G. 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 of the Fifth Working Conference on Reverse Engineering*, pages 154–163, 1998.

[Phi93] Philips Electronics N.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.

[Rai98] RainCode, Brussels, Belgium. *RainCode*, 1.07 edition, 1998. ftp://ftp.raincode.com/cobrc.ps.

[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 of the International Conference on Software Engineering*, pages 196–207, 1995.

[SV98] M.P.A. Sellink and C. Verhoef. Native patterns. In M.R. Blaha, A. Quilici, and C. Verhoef, editors, *Proceedings of the Fifth Working Conference on Reverse Engineering*, pages 89–103. IEEE Computer Society, 1998. Available at http://adam.wins.uva.nl/~x/npl/npl.html.

[SV99a] M.P.A. Sellink and C. Verhoef. An Architecture for Automated Software Maintenance. In D. Smith and S.G. Woods, editors, *Proceedings of the Seventh International Workshop on Program Comprehension*, pages 38–48, 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. In H. Yang and L. White, editors, *Proceedings of the International Conference on Software Maintenance*, 1999. Elsewhere in this volume. Available at http://adam.wins.uva.nl/~x/com/com.html.

[SV99c] M.P.A. Sellink and C. Verhoef. Scaffolding for Software Renovation. Technical Report P9904, University of Amsterdam, Programming Research Group, 1999. Available via http://adam.wins.uva.nl/~x/scaf/scaf.html.

[SV99d] M.P.A. Sellink and C. Verhoef. Towards Automated Modification of Legacy Assets. In N. Callaos, editor, *Proceedings of the Joint Third World Multiconference on Systemics, Cybernetics and Informatics and the Fifth International Conference on Information Systems Analysis and Synthesis (SCI/ISAS'99)*. International Institute of Informatics and Systemics, 1999. To appear. Available at http://adam.wins.uva.nl/~x/aml/aml.html.

[SWM97] M.D. Storey, K. Wong, and H.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.

[WOL+98] S.G. Woods, L. O'Brian, T. Lin, K. Gallagher, and A. Quilici. An Architecture for interoperable Program Understanding Tools. In S. Tilley and G. Visaggio, editors, *Proceedings of the Sixth International Workshop on Program Comprehension*, pages 54–63, 1998.