Available online at www.sciencedirect.com
SCIENCE DIRECT®
Science of Computer Programming 54 (2005) 143–211
Science of Computer Programming
www.elsevier.com/locate/scico

Architectural modifications to deployed software

A.S. Klusener[a,c,*], R. Lämmel[b,c], C. Verhoef[c]

[a]*Software Improvement Group, Muiderstraatweg 58A, 1111 PT Diemen, The Netherlands*
[b]*CWI, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands*
[c]*Department of Information Management and Software Engineering, Free University of Amsterdam, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands*

Received 5 July 2002; received in revised form 20 February 2004; accepted 17 March 2004

Available online 18 September 2004

Abstract

We discuss the nuts and bolts of industrial large-scale software modification projects. These projects become necessary when system owners of deployed systems hit architectural barriers. The mastery of such projects is key to the extension of the best-before date of business-critical software assets. Our discussion comprises the process for problem analysis, pricing and contracting for such projects, design and implementation of tools for code exploration and code modification, as well as details of service delivery. We illustrate these concerns by way of a real-world example where a deployed management information system required an invasive modification to make the system fit for future use. The chosen project is particularly suited for a complete treatise because of its size (just 90,000 LOC), and the nature of the relevant architectural modification (namely, a form of data expansion). We share the lessons that we learned in this and other architectural modification projects.

Keywords: Software malleability; Software maintenance; Definition of software architecture; Software asbestos; Software modification; Software analysis; Automated program transformation

Contents

* Corresponding address: Software Improvement Group, Muiderstraatweg 58A, 1111 PT Diemen, The Netherlands.
E-mail addresses: steven@cs.vu.nl (A.S. Klusener), ralf@cs.vu.nl (R. Lämmel), x@cs.vu.nl (C. Verhoef).

0167-6423/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2004.03.012

1. Introduction

This paper is about software systems that are already in use for a long time—five, ten or even more than 20 years. These are normally business-critical systems, which automate important business processes and processes in our society like the payment of salaries, the bookkeeping of pensions, and the payment of taxes. On the one hand, such systems require constant change to preserve or enhance the assets that are represented by these systems. On the other hand, these systems often lack malleability such that they resist to

new requirements. The mismatch between as-implemented vs. as-required systems is in no way restricted to technology or development methodologies of the past. Systems that are deployed today are the legacy of tomorrow because many of the elements of current system designs will naturally be replaced or revised in the future, e.g., APIs for data access and user interfaces, middleware technology, and model-driven technology.

The malleability attribute. This paper deals with the managed and automated modification of deployed software. Our focus is on revitalising malleability of deployed software. To this end, we realise that malleability is one of the most vital architectural quality attributes in the software world. Generally, the term malleability stands for possessing the capacity for adaptive change and the capability of being altered or controlled by outside forces or influences. In other words, the ideal software architecture for a system is one that, once the software enters its deployment phase, is continuously adaptable to changing business needs and the ever changing environment. Other names for this prominent quality attribute are *changeability*, *adaptability*, *modifiability*, *flexibility*, *maintainability*, *extensibility*, *evolvability*, and so on. We opt for malleability since this word expresses the implied attribute most accurately. Not all change is smooth; sometimes pressure is needed to adapt a system, and also this is expressed in the roots of the word malleability. The etymology of this word goes back to the Latin verb *malleare* which means to hammer. Malleability is therefore also used to express the capability of being extended or shaped by beating with a hammer or by other pressures. This image resembles the realities of the software world. The early phase of software is like melted iron, and while architecting, decisions about the grades of malleability are made for the various parts. There are immutable parts, which can never be heated up again because they are interspersed with inflammable materials. For other parts it is envisioned that they should be amenable to change when the need arises. So they are constructed in such a way that one can heat them up again to change them, but it takes considerable effort. Yet other parts have to be kept permanently in the oven to continuously blacksmith them: the *hot spots*.

Architectural modifications. Revitalisation of malleability is complementary to normal maintenance practice. That is, revitalisation of malleability requires operating at the system-wide level rather than on a per-function or per-module basis. We use the term architectural modifications to emphasise that the corresponding projects are based on the as-implemented architecture of deployed systems. While software architecture often refers to high-level designs in a way to serve the construction of new systems, our focus is instead on the modification of deployed systems. The deployed system itself (say, its source code) defines the architectural barriers that hinder smooth change. Hence, we propose the following definition of software architecture:

> *The software architecture of deployed software is determined by those aspects that are the hardest to change.*

Our definition is meanwhile adopted by others as we will clarify later. This makes us believe that we are on our way to better understand software architecture as a whole. Starting from this definition, the present paper supplies a methodology for architectural modifications. We will explain why these projects are important and special. We will

define the milestones of successful architectural modification projects: initial problem statement, iterative problem analysis, contracting, design and implementation of tools for code exploration and code modification, and finally service delivery. This is a major contribution for two reasons. Firstly, software assets are invalidated if their malleability cannot be revitalised. Secondly, architectural modification projects tend to fail, and they consume huge amounts of resources—when attempted in a naive manner.

Complete treatise of a case. We applied our methodology for architectural modifications to several real-world projects, and the paper lays out one such project that is suitable for presentation. In fact, the present paper contains the first complete treatise of a real-world project dealing successfully with an actual architectural modification to deployed software. The reported experience is meant to be instructive for carrying out future architectural modification projects. In particular, the paper presents and integrates experiences with the following subjects:

- *Problem analysis*: "How to use code exploration to learn about the modification problem? When to stop? How to get the customer involved?"
- *Cost estimation*: "How to obtain a precise and transparent cost estimation in a reasonable amount of time and with costs that are acceptable for the customer?"
- *Managerial realities*: "How to argue in favour of automated transformations as opposed to a manual approach? How to explain the complexity of the project to the system owner?"
- *Organisational realities*: "How to align offline modification at the site of the service provider with simultaneous client-site maintenance?"
- *Technological issues*: "What technology to use for modification? What amount of automation is justified for the project at hand?"

While the paper is shaped by a very tractable example project (just 90,000 LOC), we have applied the same methodology in other projects. For instance, the methodology for analysing impact, and for estimating effort and costs, was also used to provide a customer with precise information on a project where a 50 million LOC software portfolio had to be investigated for the architectural modification of extending bank account numbers to ten digits.

Organisation of the paper

- In Section 2, we introduce the real-world case that is used throughout the paper. Codename: PRODCODE project. The case is concerned with a data expansion problem in a business-critical application. This project as well as other projects will be used for discussing the realities of architectural modification projects.
- In Section 3, we substantiate that software architecture is determined by what is the hardest to change. The crux of insufficient malleability of deployed software is what we call *software asbestos*. We will then explain that architectural modifications aim at revitalising *malleability*, while using automated program transformations for removing or safely manipulating software asbestos.
- In Section 4, we describe a process for the analysis of malleability problems. This process iteratively refines an initial problem statement based on code exploration. There

are provisions that ensure a timely and precise analysis, which is meaningful to all parties involved in such projects. The resulting problem specification forms the basis for the development of automated transformation tools.

- In Section 5, we deal with project economics. Foremost, we will explain the process to agree on a cost estimation and effort distribution such that a fair contract can be signed. We will also point out the high costs and (im)possible risks of a manual approach to pursuing architectural modifications.
- In Section 6, we work out the problem specification obtained from the program analysis. All analyses and transformations are described in such detail that a basis for an implementation of automated tools and manual steps is obtained. We will employ formal rewrite rules, informal rules, and examples.
- In Section 7, we discuss tool support for code exploration and automated program transformations. We will describe the lightweight approach that was actually adopted in the PRODCODE project. We will also briefly assess the various technology options that exist, where the PRODCODE project serves as a benchmark.
- In Section 8, we conclude the paper.

2. A real-world modification example

We will now sketch the PRODCODE project. At the surface, this project was simply about extending 'product codes' from two to three digits. We will describe the problem statement of the PRODCODE project complemented by lists of challenges regarding the technical solution and service delivery. Here we will also incorporate experiences gained in other projects. By listing the challenges and the *drivers* of modification projects, we want to create awareness of the complexity of these projects, which implies that they need to be managed and that they require automated tool support. Before we introduce the PRODCODE project in some detail, we will identify project characteristics that we consider as important in the view of the paper's contribution.

2.1. Characteristics of a suitable project

Service delivery. We assume a business model with two independent parties: the *problem owner* and the *solution provider*. The problem owner is the one to pose a problem statement. It is not uncommon that the problem owner is represented via the maintenance department of a company. The solution provider takes the lead in analysing the modification problem and implementing its solution. Hence, this business model is about *service delivery*. An alternative situation is when architectural modifications are carried out by the system owners themselves, but even then different departments are involved such as the regular maintenance department vs. a task force responsible for an identified malleability problem.

Reality check. A suitable case of a modification project must concern a *business-critical* software system. Without that reality checking, we would miss crucial managerial and technological forces that drive real-world projects. One major class of business-critical systems comprises all the information systems that are deployed for processing data in the finance, insurance, or governmental sector. This implies almost certainly that our case

had to deal with a deployed *Cobol* system, which is substantiated by the following figure in [1, p. 70]:

Over 95% of finance and insurance data is processed with Cobol.

We refer to [1,69,83] for further related figures. So by opting for a Cobol case, we ensure that our approach is immediately meaningful for business-critical systems as they exist today. Nevertheless, we will soon clarify the generality of our approach and inevitability of insufficient malleability—regardless of the used language and technology.

Amenable to full treatise. On the one hand, the chosen case should be complex enough to encounter typical complications. On the other hand, the chosen project should also be simple enough to be able to convey a complete and meaningful discussion in the paper at hand. Not every project is suitable for a full treatise: sometimes the problem statement is too specialised, sometimes the types of modification are too diverse, sometimes the tooling is too sophisticated, sometimes non-disclosure limits the discussion, sometimes the system is too large, sometimes the involved languages are too unknown, and so on. Therefore, it took us some time before we came across a real-world case that is suited to be discussed as a whole. The example system in this paper, with its 90,000 LOC, is not so small that an architectural modification effort would be trivial, but it also not too large to prevent full coverage of the problem including its solution. The problem statement looks deceptively simple. The project required to expand certain kinds of fields. Data expansion is relatively well understood because it occurs frequently in practice, e.g., in the form of the 10-digit bank-account number problem. These characteristics make the case ideal for a full treatment so that one gets complete insight in a real-world architectural modification project from start to finish.

Generality. Our approach is not restricted to Cobol, nor to data expansion. It generally applies to projects that aim at the modification of deeply ingrained aspects of the as-implemented architecture of deployed systems. There are many other types of project that can (and did) benefit from our work. It is just convenient to use a project with a simple problem statement for explanatory purposes. In [78], we describe an example of another type of architectural modification effort. The paper reported on a code restructuring project for a Swiss bank, where the code was drastically changed in order to migrate from a green-screen function-key mainframe interface to a browser-like, mouse-centric PC interface. Because of the complexity of the problem, the paper had to focus on the transformation algorithm, while several project drivers could not be discussed. In [25], we address yet another type of architectural modification. The paper reported on a project for a German bank, where multiple instances of the same information system were migrated to a product line. In that paper, the focus is on the methodology of migration. Although the paper had been backed up by a real-world case, a full treatise was infeasible. For the case that we have chosen in the present paper, the problem statement can be comprehended without special domain-specific knowledge. So we can focus on what the impact of such architectural modifications comprise, how such projects are managed, and what tools can aid in cost-effective solutions.

2.2. Introduction to the PRODCODE project

PRODCODE is the codename of an architectural modification project that we carried out in 1999 for one of the global players in the finance and insurance industry. The contracting enterprise and a number of parameters of the project are made anonymous. The PRODCODE project was concerned with a deployed management information system. This business-critical system was developed in the early 1970s and is refined and enhanced to this day. The software system provides managers with important productivity summaries on their finance and insurance products. Due to various mergers and acquisitions the number of products to be monitored had grown to above one hundred, which was never anticipated. In the 1970s, a hundred financial products was an unimaginable number, so it was sufficient to identify all financial products with a two-digit product code. This two-digit type was hard-wired in the source code. So an architectural modification effort was necessary to expand the product-code fields to facilitate monitoring more than a hundred products.

Problem statement. The modification effort was initiated by the following statement:

> *Data items representing product codes have to be expanded so that the system can deal with more than a hundred product codes.*

The management information system is written in Cobol. Here is a sample declaration of a product-code field named `PRODCODE`:

```
01 PRODCODE PIC 99.
```

The picture string `99` stands for two numerical digits. To continue to deploy the management information system in the future, a new upper limit for product codes above 100 was needed. The new type `PIC 999` was considered appropriate, and hence, the above line of code has to be transformed as follows:

```
01 PRODCODE PIC 999.
```

One could argue that if two digits are expanded to three digits, it would be best to take as upper limit a 1,000 products. However, smaller values might be favourable as well in the view of a possible performance degradation. The customer requested that experiments should be carried out to find the balance between increase of product codes and performance degradation.

Summary of the system. The application consisted of 102 Cobol programs totalling 90,000 lines of code. Furthermore, 84 `COPY` books with 3039 lines of code were involved.[1] Regarding data management, a mixture of sequential and indexed files as well as DB2 tables were used. The files were accessed with Cobol's native support for file processing, and the DB2 tables were accessed using embedded SQL. There were 31 out of the 84 `COPY` books generated from DB2 tables. The code base contained 114 embedded SQL statements and 592 `COPY` statements.

[1] Cobol terminology: a `COPY` book is a kind of include file for textual inclusion; it gets expanded in place of a `COPY` statement—just like C-include files that get expanded by means of the preprocessing statement `#include`. In Cobol, `COPY` books are pervasively used to define common data declarations and reusable statement blocks.

When to make a type. This is also the title of an IEEE Software column by Martin Fowler [29], where he discusses whether or not to define types for money, currency, *product codes* and other "little objects". We view Fowler's position in this column as a support for our classification of the PRODCODE project to be regarded as an architectural modification. Namely, Fowler observes that "many architects consider such details [types for little objects such as product codes] unworthy of their attention". Fowler points out that the consequences of having not defined some type of "little objects" can be disastrous—once we need to change software. We must recover the type before we can change it. This is precisely what happened in the PRODCODE project. So types for "little objects" are indeed an architectural matter. Hence, we say that product codes are part of the software architecture because the type of product code is very hard to change. Fowler also makes this link between software modification and software architecture in another column [30], where definitions of software architecture are scrutinised.

2.3. Technical challenges

Simple problem statement, simple solution? The initial problem statement for the PRODCODE project may suggest to the reader that the problem was trivial. Thus far, the problem statement is misleading. The Y2K problem can be stated in such simple terms, too, while it is meanwhile folklore that its systematic and cost-effective solution was a major challenge. In the same way, it will turn out that the PRODCODE project was about much more than expanding fields named `PRODCODE`. We will first consider rather technical challenges, while the next section discusses project drivers that are more of a managerial or technological nature.

Impact analysis. Non-trivial modification projects of whatever kind require substantial effort regarding impact analysis. In the PRODCODE project, the assumption that relevant fields are named `PRODCODE` or alike is naive. So a more complete set of mnemonics for product codes has to be worked out. Also, identification based on mnemonics is neither guaranteed to be safe nor complete. Hence, eventually, we need a code analysis that can be used to identify relevant fields based on usage patterns. This is significantly more involved than searching for fields by name. Also, it raises the standard issue of precision. An imprecise identification of affected fields will be likely to cause a harmful modification, which is not acceptable for a system owner of a business-critical system.

It turned out that patterns other than simple data-field declarations were relevant to the PRODCODE problem. For instance, the upper limit for product codes occurs in the declaration of Cobol tables.[2] The `OCCURS 99` clause in these table declarations had to be expanded as well. Furthermore, the upper limit 99 was also hard coded in control structures. In fact, the value 100 was frequently used as an error code in the sense of "end-of-loop" or "invalid prodcode". In this context, we learned that some declarations of fields for product codes were already of type `PIC 999` as opposed to `PIC 99` just because some fields had to

[2] Cobol terminology: in Cobol, the term *table* is used for an array, say a homogeneous collection of data. In Cobol syntax, a table is declared by adding an `OCCURS` *n* clause to a field declaration, where *n* is the size of the array.

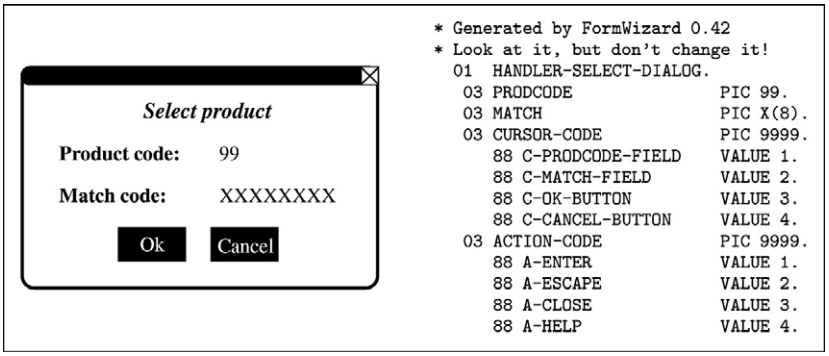


Fig. 1. A form for entering a prodcode and the corresponding data declarations that were generated by a 4GL tool for user-interface development. The Cobol application program communicates with the run-time system of the forms manager via subprogram calls where the group field `HANDLER-SELECT-DIALOG` is used for data exchange.

be able to hold the error code 100. Nevertheless the constants 99 and 100 had to be replaced systematically by the new upper limit. As we will see, there is a variety of affected patterns.

Heterogeneous system platforms. Business-critical systems normally involve a cocktail of languages, notations, and technologies. This cocktail comprises programming languages, in-house or general purpose facilities for macros and preprocessing, embedded languages for database access and transaction management, middleware technology for distribution etc., and scripting languages for gluing together programs, for controlling jobs and processes, and for configuring applications. This heterogeneity very much challenges all attempts to derive simple solutions that are evidently complete and correct. For instance, an impact analysis would need to operate across all languages. Such heterogeneous reasoning is challenging; we refer to reports on language migration [90] and dialect migration [19,42,81,98] for an indication. Heterogeneity is also the source of the 500 language problem [55], which is about the mere challenge of obtaining quality front-ends for source-code analysis and transformation.

In the PRODCODE project, we were faced with different Cobol dialects, with some standard compiler-directing statements (or preprocessing), with embedded SQL, with DB2 data models, and with generated `COPY` books for the DB2 tables. The latter issue of generated `COPY` books is an instance of the general issue that parts of the source code are generated by external tools, e.g., generators supporting interaction with 4GL languages. Consequently, one has to be careful to change the primary, proprietary sources rather than the generated ones. This is illustrated for user-interface development in Fig. 1. The `PRODCODE` field in this dialog was described in the proprietary *FormWizard* format. This 4GL code would also be subject to data expansion. We have encountered this specific problem in another project, but it was not an issue in PRODCODE project, where basic I/O was used for forms and reports.

Semantical subtleties. The solutions for modification problems are ultimately challenged by subtleties of the used programming language(s). For instance, in the PRODCODE project, we had to struggle with subtle rules for conversion between Cobol's types for

numeric and alphanumeric data. In other modification projects, we have encountered complications as follows:

- Semantical subtleties of exception handling.
- Semantical subtleties of object destruction (cf. memory leaks).
- Precision issues for cross-language data conversions.
- Undocumented or obscure API uses, e.g., error codes for file handling.
- Fragile race conditions in distributed or multi-user code.

Such subtleties are the source of unsound modifications that look seemingly correct. Also, these subtleties are likely to lead to complex and defensive schemes of modification.

2.4. Project drivers

We have substantiated that large-scale modification projects are challenging as far as a proper technical solution is concerned. However, one should not overestimate the specific technical problem in a project. In our experience, modification projects are *driven* by other factors. These project drivers are normally related to the economical, managerial, legal, and technological characteristics of both problem owner and solution provider. It is important to identify such drivers upfront in a modification project, in fact, before contract signature, because they affect the costs substantially up to the level of infeasibility of the project as such.

The light-switch myth. At an early stage of modification projects, customers tend to think of a well documented, user-friendly, mature tool that could be used by maintenance staff to solve all problems automatically, transparently, and conveniently on their site. The ideal tool has the complexity comparable to a light-switch: just turn the handle and the result is a neatly converted program. In the PRODCODE project, we rapidly realised that the complexity of the problem required a semi-automatic approach. The overhead to build tools that could be used by the maintenance programmers would be unaffordable. Furthermore, the software and hardware architecture at the client site and our site turned out to be largely different. Our exploration and transformation tools are currently Unix based, whereas the client's software, including the PRODCODE application, ran on a mainframe with a different operating system. A Unix to mainframe conversion of the modification tools could not have been paid off by the project at hand. Consequently, we had to communicate with the customer to agree on the more realistic option for service delivery, where the conversion is carried out by us on our site.

Syntax retention. This non-functional requirement for tool support means that the changed programs should be identical to the original sources in all locations that did not require modification. This normally includes preservation of syntactical patterns, lexical details, spacing, and comments. Syntax retention is a prerequisite for using lexical technology for the line-by-line inspection of the differences between the original and the new system. Without syntax retention, the customer cannot inspect differences efficiently because too many locations will be notified, even though their appearance was changed only. Customers will not necessarily come up with this requirement themselves. This might be implied by the primary way of thinking of modification: modification in terms of hand-coding. Then, one does not realise that automated conversions could possibly perform pervasive

code changes such as stripping off line numbers, lexical or syntactical normalisations, and pretty printing. In fact, syntax retention and similar non-functional requirements challenge the implementation of transformation tools. In the PRODCODE project, we anticipated the need for syntax retention.

Limited testing opportunities. In the PRODCODE project, regression testing of the software and testing that the new requirements are met was to be done by the customer. At first sight, this set-up appears to reduce our effort, but it soon becomes clear that this set-up puts stress on the process as to be able to guarantee a correct solution even without testing. Our customer would have found it unacceptable to go through several iterations of modification and testing—be it just to compensate for our omissions. Also, we cannot expect the customer to carry out advanced testing for the new requirement. Generally, we cannot even expect the owner of deployed software to be in the possession of a serious harness for regression testing. In conclusion, modifications must be "obviously" correct so that a modified system does not produce unforeseen (i.e., undocumented) problems at the client site. This is a strong incentive for using automated transformations. Also, automated checks are to be performed on both the converted system and the transformation rules themselves. However, we are not proposing a complete, formal verification of architectural modifications because these projects are normally not amenable to such a treatise.

Scattered modification projects. Modification projects naturally involve the maintenance departments at the site of the system owner even though the core of the automated modification might be delivered by an external solution provider. Such scattered modification projects imply extra effort for synchronisation. In the PRODCODE project, the customer did not want to outsource the database part of the expansion, and we had in fact no access to the 4GL code. Also, for reasons of confidentiality, the maintenance department was responsible for migrating all the indexed files and DB2 tables. This implied the challenge of setting up a process that guarantees consistent changes. Recall that the Cobol code included several `COPY` books that were generated from the DB2 tables. In order to make our part of the expansion as complete and reliable as possible, we generated a report for the client to point out affected database columns that needed expansion at the client site.

Dead code. In the course of modification projects, one often encounters code patterns that are difficult to comprehend and difficult to handle. Often these patterns are symptoms of dead code. Identifying abnormal patterns is beneficial because it prevents us from trying to cover these patterns. However, the approval of dead (or dangerous) code can be involved as well since simple static checks are too conservative, but additional knowledge about program usage and data are often required. In the PRODCODE project, we found suspicious tables that were indexed by product codes, while they had fewer than 99 entries. It eventually turned out that the size of these tables reflected an earlier upper limit for product codes, and the code did not trigger problems because it happened to be dead. Business-critical systems are likely to contain dead code of significant quantity. Many updates were performed on these systems, and this can lead to as much as 30% of the actual volume of the code being dead [36]. Often entire programs are dead in the sense that they are not used anymore.

Off-loading and shipping back. In many organisations there is no predefined process to deliver complete systems to external sites. Proper institutionalisation of configuration and version management requires level 2 of SEI's capability maturity model (CMM), while 75% of all companies are at level 1 [38, p. 30]. As a consequence, the original system might initially come in batches, where some files might never get shipped, other files are shipped more than once—maybe in different versions. This is one example of difficulties of integrating an external large-scale software modification effort into the client's everyday practice. Such aspects can hamper swift throughput of the project, and even cause failure. Another problem to prepare for is that once the software is off site, the client will change the system in the meantime. Once we have done the modification work for a (possibly incomplete and by then outdated) version of the system, the next phase is to manage a separate shipping project with the client so that the complete, most recent version of the system is shipped once again and modified with the tools. Then we can apply the tools to the system, and ship the new version say the next day. The client needs to freeze other maintenance activities during this period.

There exist more project drivers, but they will differ from project to project. The drivers in the above list occur often and they influence the duration and costs of projects significantly.

3. Software asbestos

Throughout its life-cycle, software becomes invaded by incidental or accidental issues just as buildings have been unintentionally invaded with the then unknown characteristics of asbestos. We use the term *software asbestos* to refer to the implementation of unintentionally immutable parts of a software system that severely hamper anyone making necessary changes. Architectural modifications are precisely meant to remove or safely manipulate software asbestos, e.g., the hard-coded product codes in the PRODCODE project. The following matrix places architectural modifications in a context:

Change	*easy*	*hard*
anticipated	maintenance (best practice)	design while alter (malpractice)
unanticipated	maintenance (sheer luck)	architectural modification (life-cycle enabling)

That is, architectural modifications are about required changes that had not been anticipated, while they happen to be difficult because of asbestos. Architectural modifications go beyond regular maintenance activities, which may be concerned with modifying single subprograms, or systematically revising all calls to a given subprogram, or wrapping a subprogram (cf. *anticipated, easy changes* in the above table). Architectural modifications are rather concerned with deeply ingrained aspects of the as-implemented architecture of a deployed system.

We note that the inevitability of software asbestos and the continued need to keep business-critical systems malleable is directly linked to some of Lehman's laws of software evolution [5,59,60].[3] All possible forms of software asbestos are concerned with choices, abstractions, and idioms in software development:

> *choice for implementation languages, platforms, development environments, libraries, domain-specific APIs, proprietary communication protocols, choice for partitioning into components, directory structures, CPU partitioning, layout conventions, variable naming schemes, use of preprocessing, middleware for distribution, synchronisation aspects, the security policy, ...*

We will now explain why software asbestos exists in the PRODCODE application and why software asbestos is generally inevitable even if we could dream up the technology and methodology for business-critical systems. In due course, we will encounter several forms of software asbestos. We also discuss the issue what to do with a contaminated system: modify it, replace it, keep it as is, or otherwise. In the final part of this section, we can then fully substantiate the link between software architecture and (architectural) modifications of deployed software.

3.1. In Cobol's defense

The primary challenge regarding a technical solution for the PRODCODE problem is that fields for product codes first need to be identified, since these fields are not readily tagged in any way. Some fields of type `PIC 99` will be product codes; others are not. We learned that several different types are used—not just `PIC 99`, but also `PIC XX`, `PIC 999`, and others. In fact, 12 different types were used for same concept: a product code. Given the 90,000 lines of the PRODCODE system, this implies a different type for every 8,000 LOC. Further complications are related to hard-coded literals, e.g., 100, which serves as an error code. Now it is legitimate to raise the following questions:

- How can someone implement an issue that looks so simple as the concept of product codes in such a complex and unfathomable manner? Is this an exception—a freak accident, or is this type of implementation common practice?
- The problem would be simpler if not trivial if declarations of fields for product codes would refer to a designated type. Also, constant declarations could have been used for error codes, special values, and so on. Why did this not happen?

In replying to these questions, we want to provide evidence that (Cobol-based) business-critical information systems (such as the PRODCODE application) are not the result of careless programming by careless programmers incapable of adhering to the simplest best practices in software engineering. To this end, we need to analyse Cobol idioms.

Hard-coded types and literals. Cobol did not permit a reusable declaration of a type for product codes. This immediately rules out a malleable software architecture in which the type for product codes amounts to a hot spot. Instead, one was encouraged to use

[3] In particular: law I, continuing change, law II, increasing complexity, law VII, declining quality.

hard-coded types. In so far Cobol is (was) inferior compared to languages like Pascal, C, Modula, Ada, and typed OO languages. Similarly, the error code 100 was hard-coded because the proper declaration of constants was not permitted either. The use of several types of product code as opposed to merely `PIC 99` (or `PIC 999`) is related to common Cobol programming practice to heavily use formatted types for reports and screen I/O, but it is further encouraged by Cobol's weak type system.

A reference encoding. Meanwhile, with a Cobol 2002 compliant implementation, the type for product codes could be declared once and for all in the `DATA DIVISION` of each program, or even in a `COPY` book for system-wide use:

```
01    PRODCODE-TEMPLATE    IS TYPEDEF.
 03   PRODCODE             PIC 999.
      88 PROD-ERROR-CODE   VALUE 100.
```

This declaration uses Cobol's new `TYPEDEF` clause for type declarations [18,34,74]. The `TYPEDEF` clause indicates that `PRODCODE-TEMPLATE` is a template, i.e., type declaration, and not a normal data field which the compiler has to allocate space for. As a result, the picture string `999` no longer needs to be scattered over the normal field declarations. We should note that we immediately favour the type `PIC 999` over the type `PIC 99` here because we uniformly assume fields that are able to store the error code 100, or even a greater value for the converted PRODCODE system. Indeed, the type declaration declares a condition name `PROD-ERROR-CODE` for the error code 100 once and for all. One can refer to such a template with the `TYPE`-clause. To give an example, we construct a group field `OUTPUT-DATA`, which includes a field for a product code:

```
01    OUTPUT-DATA.
 03   OUTPUT-HEADER     PIC X(42).
 03   OUTPUT-PRODCODE   IS TYPE PRODCODE-TEMPLATE.
```

For clarity, this code is equivalent to the following "expanded" code:

```
01    OUTPUT-DATA.
 03   OUTPUT-HEADER     PIC X(42).
 03   OUTPUT-PRODCODE   PIC 999.
      88 PROD-ERROR-CODE VALUE 100.
```

These new constructs are clearly convenient to encode a reusable type of product code, and hard-coded error codes are avoided as well. Type declarations do *not* make complications go away related to the different types that are used for product codes. That is, code for conversion between the types is still scattered throughout the system, e.g., in the form of implicit conversions in `MOVE` statements. One can imagine defining abstract data types for product codes including conversion routines, which would be possible using Cobol's subprograms or the object-oriented concepts of contemporary Cobol [52].

Idioms available at that time. Since type declarations were not available at the time, when most Cobol-based, business-critical systems were built, one might wonder what other idioms could possibly have been used to avoid hard-coded types and literals. We will now discuss all such idioms.

Data-field declarations can be augmented by condition names. These condition names are like constants because they can be used in conditions in IF statements instead of "PRODCODE = 100", and in assignments using the SET statement. This idiom is insufficient because condition names had to be *repeated* for each new product-code field. We would really need type declarations to avoid this, as illustrated with the PRODCODE-TEMPLATE above.

The ENVIRONMENT DIVISION of a Cobol program is used for various kinds of declarations. There are also forms that come close to constant declarations. One can define character sets with the ALPHABET phrase, and enumerations of literals with the CLASS phrase. In turns out that both phrases are too restricted to declare numeric constants like 100 for general use in a program. Also, there is no form of declaration that comes even close to type declarations.

Some compiler vendors have made extensions to their Cobol implementations to solve the hard-coded literal problem. MicroFocus Cobol has been supporting symbolic constant declarations for several years. For this purpose, the level 78 for data declarations is used. Fujitsu Cobol supports a declaration form for so-called SYMBOLIC CONSTANTS that is part of the SPECIAL-NAMES paragraph. This is an indication that indeed the pervasive problem of hard-coded literals is not easily solved using the existing palette of constructions in Cobol. Hard-coded types can still not be eradicated in this manner.

We could attempt to place a reusable declaration of product codes in a COPY book. Several elements of the declaration should be variable then: the level number 01, 02, 03, ..., 49;[4] the name for the field; and the condition name for the error code 100. To this end, we assume a COPY book "PRODCODE.CPY" of the following trivial form:

```
LEVEL   DATA-NAME       PIC 999.
        88 ERROR-CODE   VALUE 100.
```

We can now attempt to reuse this declaration by means of a sophisticated COPY statement that replaces LEVEL, DATA-NAME, and ERROR-CODE. We reconstruct the earlier example for using type declarations:

```
01    OUTPUT-DATA.
 03   OUTPUT-HEADER PIC X(42).
 COPY "PRODCODE.CPY" REPLACING
   ==LEVEL==        BY 03
   ==DATA-NAME==    BY OUTPUT-PRODCODE
   ==ERROR-CODE==   BY OUTPUT-PC-ERROR.
```

The REPLACING mechanism is purely textual, i.e., COPY books are inlined during preprocessing. This low-level idiom is indeed not used in practice. Also, the assumption of a uniform type PIC 999 further disqualifies this approach because at that time, when the system was deployed, this type would have been considered wasteful.

None of the above idioms can be considered a proper solution.

[4] Cobol terminology: the order of level numbers expresses grouping of fields. That is, a greater level number indicates nesting relative to the previous field, while the same level number for contiguous fields means that the fields are part of the same group.

3.2. Universal inevitability of asbestos

The Cobol-biased discussion revealed that existing Cobol-based business-critical systems could not possibly avoid software asbestos related to hard-coded types for product codes and others. Consequently, if we want to prolong the life of these software assets then we have to engage into an architectural modification effort. This conclusion is of dramatic impact because 95% of finance and insurance data is processed with Cobol [1, p. 70]. However, the celebrated focus on hard-coded types in Cobol needs to be complemented by the following two key observations:

- The mere existence of type declarations or abstract data types in a language of choice does not guarantee that these idioms are consistently used such that malleable system architectures will be implied.
- Hard-coded types constitute just one particular form of software asbestos. Any choice of language, platform, environment, style, etc. has its own collection of examples of software asbestos.

As we will substantiate, the bottom line is this: no matter how hard one tries, any software system is likely to be contaminated with several forms of software asbestos. It does not matter so much whether the asbestos is a result of deprecated idioms or styles, or whether a software architect failed to anticipate a certain dimension of malleability. Software asbestos is a fact of life.

Too many types, too little time. Having support for type declarations, and using this idiom widely is not a guarantee for success. To explain this, we resort to another language since we have not seen any Cobol-based business-critical systems that uses type declarations of the new age. We discuss type declaration problems for a language that had type declarations from the start: ABAP/4 (short for advanced business application programming). This is a 4GL based on Cobol with support for type declarations. In fact, ABAP/4 is the language for the implementation of the SAP software, which certainly counts as business critical. In [88], Spitta and Werner lay out an analysis where the reuse of the data types in the implementation of SAP R/3 was analysed. The system is huge, 40,000 programs, 34,000 functional modules, and 11,500 tables, and therefore an excellent candidate for data type reuse. Spitta's and Werner's analysis revealed that the majority of data type declarations in this huge system were not reused at all, namely 69%. Also, 6.2% of the data type declarations were not used at all. And of the 15.6% that was reused, the reuse was restricted to two to five times; the remaining 9.2% was apparently reused more often. These results indicate that it is rather difficult to maintain a precise set of types in larger applications. Throughout the life-cycle of software, types do emerge, and it is not always clear to the developers which types already exist, or how certain types are related. This knowledge is not explicit in software. At a certain moment, variations of existing types are created although they are conceptually equivalent to existing ones. This leads to software asbestos.

Asbestos related to APIs. In the last 10 years or so, development platforms have started to rely heavily on APIs. While a conservative Cobol program uses Cobol's file-processing constructs, a Java application uses JDBC for database access, and a .NET application uses the ADO interface for database access. Even the Cobol code of the PRODCODE application

uses an API for embedded SQL. All the existing business-critical code employs a myriads of general-purpose, domain-specific, or home-grown APIs, and often there exist several versions of them. While the benefits of APIs are beyond dispute, pervasive API usage is clearly a prime cause of software asbestos. This status becomes evident whenever we want to migrate from one API to another, or maybe just to another version of the same API. To give an example, here is some fairly simple C++ code which creates a "Main Window" in terms of the (pre-.NET) Windows API:

```
HWND hwndMain = CreateWindowEx(
  0, "MainWinClass", "Main Window",
  WS_OVERLAPPEDWINDOW | WS_HSCROLL | WS_VSCROLL,
  CW_USEDDEFAULT, CW_USEDEFAULT,
  CW_USEDDEFAULT, CW_USEDEFAULT,
  (HWND)NULL, (HMENU)NULL, hInstance, NULL);
ShowWindow(hwndMain, SW_SHOWDEFAULT);
UpdateWindow(hwndMain);
```

Mapping this code to Windows Forms in the .NET Framework results in the following:

```
Form form = new Form();
form.Text = "Main Window";
form.Show();
```

This looks relatively simple. As with the idea of mapping `PIC 99` to `PIC 999`, we should not miss the hard problem of API migration. That is, we first have to locate the slices of code that amount to a certain idiom of API usage, and then to define a representation of this idiom using the new API. To make this a bit more concrete, consider the configuration of the main window in the original code snippet. If we always knew to find a Boolean expression like `WS_OVERLAPPEDWINDOW | ...`, then this configuration would be sufficiently explicit for conversion. However, in actual code, this expression could be arbitrarily formed, or we could need to resolve variable references, or it might even be computed dynamically. This is just a very simple example, but it indicates that API conversion is a challenging form of architectural modification.

Asbestos related to preprocessing & Co. Mechanisms for preprocessing and macros are specifically meant to make software more malleable. This is not a new idea. For instance, Peter Brown wrote in the 1970s about the use of macro-processors to construct portable software [16] using conditional compilation for targeting different platforms. Portability is clearly a form of malleability. The PRODCODE developers could have been using preprocessing to simulate type and/or constant declarations as required for PRODCODE fields. Cobol 2002 supports preprocessing statements these days [17,34], but we could also have been using the C preprocessor `cpp` [41] (even for Cobol), the traditional Unix tool `m4` [73], a reuse-oriented preprocessor [3], or a home-grown preprocessor. For instance, constants would be defined with a `#define`-like statement as opposed to the hard-coded constants in the PRODCODE application. Encoding reusable type declarations is a bit more involved in that we need to come up with suitably parameterised macros.

Preprocessing has not been identified in studies on best practices in software engineering [38,64,65]. On the contrary, the assumed benefits did not show up at that

time. Elsewhere, various authors [4,26,27,61,86,87] report on the implications of using preprocessors like `cpp` and `m4` for portability and code reuse reasons. The executive summary of their reports is the following:

> *Using preprocessors to make software more flexible, tends to lead to unmaintainable source code in the end. The presence of `cpp`-like constructs in programs is a headache not just for maintainers, but also for developers of tools for program comprehension and others.*

So the use of these mechanisms makes software less malleable in a certain dimension. Hence, the use of preprocessing and macros introduces some new forms of software asbestos. In some cases, this necessitates architectural modifications that eradicate conditional compilation constructs and others. For instance, from people at Philips Research we learned that they had written a so-called `#if-def`-resolver, to shed light on the software in television sets [68]. Tool support for the eradication of preprocessing statements is also discussed in [4].

Model driven architecture—a silver bullet? In the last few years, the object management group (OMG) has developed the model driven architecture (MDA) approach to building platform-independent applications [43,67]. This approach relies on OMG standards such as UML, XMI, and others. MDA promises that applications can be realised on different open or proprietary platforms, including Corba, J2EE, .NET, and Web Services. The key requirement is that the developer differentiates a platform-independent model (PIM) and a platform-specific model (PSM) of the developed software. The MDA approach aims at supporting the derivation of the PSM from the PIM. At first sight, the MDA approach can be viewed as a major attack against software asbestos coming in the form of scattered platform-dependent code. We quote Martin Fowler [31] to assess the eternal independence of the MDA approach:

> *When MDA talks about platform independence, it's treating your programming environment as the platform. But this is complete hogwash. MDA uses a bunch of OMG standards (UML, MOF, XMI, CWM etc.), these standards are every bit as much a platform as the Java stack (or the .NET stack for that matter). All you are doing is swapping one (hardware/OS) platform independent programming environment for another. You aren't getting any more independence.*

So using the MDA approach is not likely to make the problem of software asbestos go away. The pervasive use of complex standards underlying MDA and the specific mechanisms for the derivation of PSMs are good candidates for software asbestos. A difference is perhaps that we will also be faced with asbestos in *models* in addition to just asbestos in source code before.

3.3. The future of contaminated systems

Given a contaminated system, there are different perspectives for the system:

- *Conservation*: As long as the asbestos is not in the way, as long as we do not suffer from the associated lack of malleability, we are likely to keep the system as is. For instance,

in the PRODCODE application reside many hard-coded types other than for product codes. We will not invest in type recovery for these.

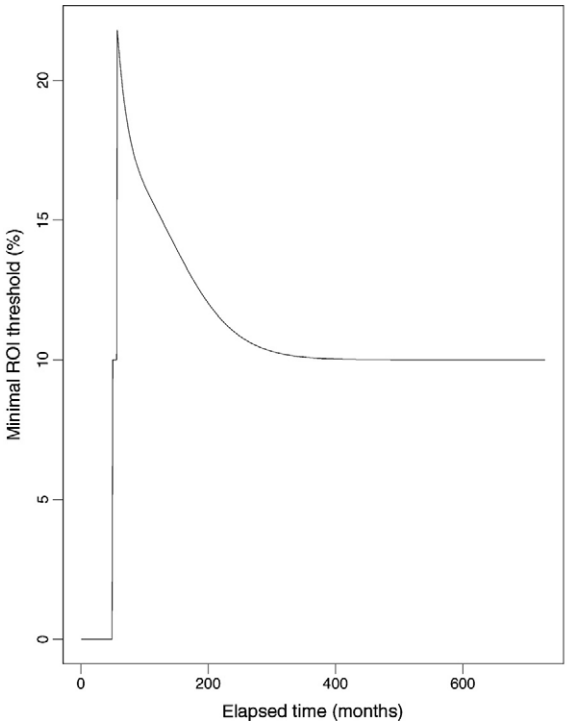
- *Modification*: The PRODCODE project exemplifies that a problem with software asbestos is normally addressed because system owners are faced with insufficient malleability. In order to enable a specific enhancement of the system, an architectural modification project must be embarked on. We also call this life-cycle enabling.
- *Preventive modification*: In some cases, a preventive architectural modification can be preferable, if the associated benefits are considered worth the effort. Such benefits typically include improved program comprehension and cost reduction of normal maintenance activities.
- *Replacement*: Rather than revitalising malleability, system owners could possibly abandon their deployed systems, and replace it by a new system that includes previous services but also new ones that were hard to accomplish within the as-implemented architecture of the deployed system.
- *Starvation*: System owners could also decide to keep the system as is accepting the negative impact of not being able to eradicate deficiencies or to add a valuable service. Reasons for such starvation include unaffordable costs for revitalisation or replacement, and the foreseeable end of the life of the business-critical system.

We will now discuss these potential perspectives in some more detail.

Modification vs. replacement. At first sight, replacement seems attractive: insufficient malleability of a deployed system could serve as a good reason to develop a new, modern system. However, in most cases such replacement cannot be motivated in economical terms. The large investment for a deployed system has normally be debited in the past. Hence, the deployed system only has operational costs. A new system requires a large budget to rebuild the current one. This also includes high operational costs during the first years, where the "unknown" features of the current system are recovered, by the virtue of failure of the new system. These costs can never be compensated by lower operational costs of the new system for the rest of the system's lifetime. In Fig. 2, we illustrate how an IT investment stresses the return on investment (ROI) threshold. Even when the requirements change drastically, i.e., when a significant budget is needed to adapt the deployed system, the option for incremental changes comes with less risks at the very least. Our arguments assume that system owners must be able to perform architectural modifications in a managed manner. Otherwise, starvation is the only remaining option.

The mere existence of software asbestos is normally not a sound reason for replacement since the costs and risks are simply too high. Replacement is sometimes implied by strategic or political forces such as the replacement of home-made systems by standard (ERP-)packages.

Modification: automated vs. manual. Our structured process for carrying out architectural modifications employs automated program transformations. In current IT practice, the inability to update software in a structured and automated manner implies that most modification projects, where significant amounts of the system need to be changed, are not performed in the first place. If large architectural modification projects are performed in a more or less manual manner, then they are likely to fail. Namely,



Suppose that after 50 months of development, a 10% net return is required to materialise. Because of ongoing development ending in month 77 and high operational costs for the system in the first years, the actual ROI threshold must be significantly higher. That is, the visible peak indicates ROI pressure resulting from costs for development and operation. By contrast, continuous operation of a deployed system including incremental system adaptation does not lead to such ROI pressure.

Fig. 2. A quaver curve for the lower-bound ROI threshold to achieve 10% net return in a software development project. The figure is adopted from [92].

a manual approach is very expensive and error prone. A manual approach leads to patchwork incrementalism, resulting in short-term extensions of the system. The lack of automation in software modification projects is only one factor. In addition, large-scale projects often suffer from inappropriate project management. These projects often have harsh deadlines, and there is little commitment from management since the business sees no added value. Projects like the described PRODCODE project are launched when management realises the benefits of an automated approach to performing large-scale architectural changes. To this end, we compile an open-ended list of reasons in favour of automation:

- The bottom line for using transformations is that one keeps control over the situation: one knows exactly what needs change, what is changed, how it is changed, in which order, and how to change the changes themselves. Automated transformations are a scalable means to deal with large code bases. Automation of modifications contributes to a structured process.
- People are not good at consistently applying rules by hand over and over again. If there are several variants of modifications, this is even more error prone. Using automated program transformations for each and every change solves this issue. Moreover, for large code volumes, manual approaches are strongly discouraged by Gartner Group: for

systems that comprise more than 2 million lines of code, manual mass-update projects could be considered professional malpractice [32,37].

- In an automated approach, most work is about the exact formulation of the problem including all its variances—as we will demonstrate for the PRODCODE project. The actual implementation of the problem specification is then normally a low effort with respect to a manual approach. In case of a manual approach, complications and variances are often overlooked in the initial phases, which causes problems and delays when the complications and variances finally are encountered.
- Changes often have to be performed on different versions of a system, e.g., the modification rules are developed using an initial snapshot, while they need to be applied to the production system later. Also, source-code portions of the system often occur late. Hence, one needs an automated process to handle all batches and all versions uniformly. Even manual changes are recorded (using for example the Unix tool `diff`) such that they can be included as patches in this automated process.
- The mass change needs to be executed in a short period of time, when normal maintenance is put to hold. For some aspects such as data migration, it might even be necessary to take the running business-critical system offline. The implementation of an automated process may take a while including recording manual steps, but the final application of an automated process can normally be limited to one night or one weekend.
- If more than one version of the system is running at remote locations, one must perform the mass update several times. Also, converted systems can display unexpected side-effects, for instance in connection with other systems. Then these systems need an update as well, and the conversion needs to be repeated. Automated transformations provide a high degree of reuse in such situations.
- Variants of a new system are often required to compare certain aspects for these variants. Using automated transformations, these variants can be generated easily. In the PRODCODE project, this was the ability to do a performance analysis for the new upper limit between `100` and `1000` for product codes. Transformations can also be of use to temporarily include code that facilitates assertion checking and debugging.

A fully automatic solution is not always feasible, and it is sometimes not cost effective. For instance, a modification problem that involves heuristics to determine affected parts of the system often necessitates interactive steps for approval by maintenance programmers. In an extreme case, the automation could be restricted to the generation of a report, which is then applied by maintenance staff in a manual manner. To this end, special interactive tool support can be provided such that programmers basically walk through the generated report and navigate to the affected code locations without ado. Similarly, there is a tension between handling less frequent or highly complex idioms by specific, manual changes per occurrence rather than providing a general rule for the underlying code pattern(s). The decision how much automation is necessary and whether generic modification rules are required has to be made while relating to the technical analysis of the problem at hand, and to the drivers for the project.

Asbestos treatment. In an endeavour to deal with software asbestos in architectural modification efforts, there are three different options:

- *Safe change*: We trace the software asbestos, and adapt all affected code patterns. This is precisely what we have done in the PRODCODE project. A similar effort is described in [76], where hard-coded SQL error codes were updated with the new codes for a new version of the database management system.
- *Removal*: We trace the software asbestos, and remove it entirely. In the case of the PRODCODE problem, this would mean turning hard-coded literals into symbolic constants, and replacing hard-coded types by type declarations. This option requires a Cobol 2002-compliant compiler, or some extensions at least.
- *Workaround*: We trace the software asbestos, and supply a workaround. A typical example is using windowing to deal with a Y2K problem. That is, the hard-coded types for dates with 2 digits for years are preserved, but the workaround interprets the years 00–99 differently.
- *Hybrid*: These three options can also be mixed. For instance, different degrees of asbestos removal were used in Y2K projects. While it was not uncommon to factor out date arithmetic into designated routines, the use of hard-coded types was normally not eradicated.

Note that we always start with tracing asbestos prior to an asbestos-aware transformation. Hence, we obtain a stricter separation of tracing and transformation, if we assume that the result of tracing is documented temporarily or permanently in the source code. In fact, this is a form of scaffolding [48,77], say a structured comment. This is a useful method to drive a transformation, or to repeatedly operate on the asbestos. Scaffolds are also useful for interactive transformations, where the scaffolds mark the relevant spots to be affected, and they encode options for user intervention.

3.4. A definition of software architecture

We want to utterly clarify the architectural dimension of the presented work. To this end, we discuss definitions of software architecture, and we argue that our view, which focuses on the modification of deployed software, is valuable.

Pluralism in software architecture. There exist many proposals for definitions. For an extensive list of definitions we refer to [85]. Most definitions seem to suggest, in more concrete terms, that software architecture is about certain views on the system design. A fairly established definition by Bass et al. [2] is given as an example:

> *The software architecture of a program or computing system is the structure of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.*

This style of operational definition is widely used, where the definition provides a list of issues such as components, their properties, relations between the components, and others. It also seems that many definitions tend to share some aspirations with civil architecture in the sense of focusing on integrity of construction. In *The art of systems architecting*, the idea of architecture is perceived as follows: “Models are used to control the construction process and to ensure the integrity of the architectural concept” [62, p. 140].

The architecture of deployed systems. In the case of deployed systems, requirements or design documents are normally outdated and often even non-existing. Also, the

original designers and programmers are no longer available to reflect on the architecture. Furthermore, successive changes have blurred many original design decisions. Moreover, there is no guarantee whatsoever that the encountered barriers were considered at all in the original architectural design. This leads us to conclude that deployed systems themselves define an *as-implemented* architecture. There is a body of research on architectural recovery of systems [23,33], but again, the recovered architectural information is of a kind we just have discussed: high-level views on the system design. We would rather want to focus on malleability of deployed systems.

Who needs an architect? In an IEEE Software column of this name, Martin Fowler and Ralph Johnson (the latter by means of quotations) discuss the role of an architect, and soon they discuss defining software architecture [30]. They criticise phrases like "architecture of a software system ... is its structure of significant components" because these phrases miss the relativity of significance. Significant for whom? For customers? Johnson votes for expert developers, and he offers a definition of software architecture:

> *In most successful software projects, the expert developers working on that project have a shared understanding of the system design. This shared understanding is called architecture.*

So architecture is about important design issues, important for the expert developers. And the architect is the person who worries about such important things. Johnson also identifies another style of definition such as in "architecture is the set of design decisions that must be made early in the project". Johnson proposes that this should actually read as:

> *Architecture is the set of design decisions that you wish you could get right early in a project, but that you are not necessarily more likely to get them right than any other.*

Fowler picks up this definition. He simply wonders why people would feel the need to get some things right early in the project. This is Fowler's answer including his implied definition of software architecture:

> *The answer, of course, is because they perceive those things as hard to change. So you might end up defining architecture as things that people perceive as hard to change.*

While this definition is somewhat oriented towards development or construction of software, its emphasis on changeability (or malleability) is 100% in line with our architectural view on deployed software. Fowler uses the wording that architecture is what people "perceive as hard to change" because it depends on the technical and imaginable abilities of the architect if he or she can arrange for something being easy to change. This is the reason for perceivability being part of the definition.

As-implemented architecture. It is only a little step from Fowler's to our definition. When modifying deployed systems, we are faced with an as-implemented architecture. The detailed design decisions that would have made it easy or hard to change aspects belong to the past. There is no longer any point about perceivability, while from Fowler's perspective there is. Also, the architecture is partly incidental or accidental. This motivates the following definition:

The software architecture of deployed software is determined by those aspects that are the hardest to change.

Whether something is hard to change or not is judged by the standards of software development and maintenance practice. Managed and automated architectural modifications are precisely meant to enable changes that would otherwise be too hard. To conclude, software architecture is about the immutable aspects of a software system. It really needs architectural modifications to replace these parts. Candidates for such immutable aspects are the chosen implementation languages, platforms, development environments, predefined libraries and APIs, proprietary communication protocols, database schemas, and so on. However, not every instance of such aspects counts as architectural. Whether a given aspect in a given system actually counts as architectural solely depends on how difficult it is to change the encountered implementation of the aspect.

Bibliography. We started to use this definition of software architecture three years ago, and at that time it deviated from the status quo. Meanwhile, the definition has been adopted or independently discovered by others. For instance, Andrey Terekhov adopted our definition on his slides presented at the Dagstuhl seminar 03061 Software Architecture [89]: "We believe that the best definition of software architecture is the following: Software architecture is comprised of all features that are difficult to change in a system". At the same seminar, Jan Bosch, independently of us, develops a very similar characterisation on his slides [8]: "Software architecture is hard to change.... [it] is [the] static, the stable part of the system.... Inflexible architecture is a fact of life!". Gert Florijn refers to our definition in his course notes on software architecture [28] as a good example of a definition for software architecture. We also recall the IEEE Software columns by Fowler [29,30], which are aligned with our view on software architecture. This makes us believe that our alternative definition is of general use.

The next sections will describe an approach to carrying out architectural modifications.

4. Analysis of modification problems

We will now define a process for analysing the actual problem in an architectural modification project. The result of this analysis is a problem specification, which forms the basis for an estimation of effort and costs. We will focus here on the analysis of the technical solution, where we recall that a complete understanding of a modification project also requires the identification of project drivers—as listed in Section 2.4.

Disclaimer. Some readers, who are very experienced in the area of data expansion, might feel that our approach to come could be more sophisticated. It is the case, indeed, that some companies are in the possession of advanced technology and methodology to endeavour expansion projects. We would like to stress that data expansion serves as an example in this paper. That is, we deal with all the more or less well known issues in doing data expansion. We aim at the illustration of a reusable method and a general process for architectural modifications, rather than the description of the ultimate approach to data expansion.

4.1. The process for problem analysis

We use the following process for problem analysis:

(1) **Explore**: We use code exploration to learn about the problem. This will lead to the substantiation of assumptions about affected patterns, variation points, and a quantitative measure.
(2) **Model**: We sketch an operational model that approximates the technical solution of the identified part of the problem. This concerns analyses to determine affected patterns, and transformations to adapt the code.
(3) **Estimate**: Based on the operational model, knowledge about project drivers, and quantitative measures, we estimate the effort needed to actually solve the identified part of the problem.
(4) **Review**: We reflect on the operational model gathered so far as to identify drastic sources of incompleteness. This includes the exposure of undue assumptions about programming conventions and encoding idioms adhered to.
(5) **Discuss**: The findings are discussed with the customer's domain experts. This is meant to ensure a feedback cycle so that we can take advantage of the customer's insights, and to create awareness of identified complications at the same time.
(6) **Loop**: Steps 1–5 are repeated until the risks and problem variances are well understood such that the model has sufficient detail for cost estimation.

The purpose of this process is twofold. Firstly, the amount of affected code and the diversity of relevant code patterns are estimated. Secondly, it is superficially investigated how to make the changes. The customer's participation ensures that the impact and the results of the analysis are meaningful to the customer. In the PRODCODE project, the maintenance department at the client site was loosely involved in the problem analysis. In addition to daily meetings, there was a contact person who could be consulted by us for immediate help.

4.2. The initial problem statement

We will now enter the above process starting from the initial problem statement for the PRODCODE project. Remember this very succinct statement: The system is required to work with more than a hundred product codes. We will now go through the phases *explore*, *model*, and *estimate*.

Code exploration. The customer told us that the names of fields for product codes normally contain the string `PRODCODE`. So we decided to make a rough estimate on how many affected data declarations were present in the system. To this end, we employed the simple UNIX command `grep` to list all declarations for data fields containing the string `PRODCODE`:[5]

[5] In this paper, we use `grep` under SUN Solaris/SunOS 5.8. Our commands may need minor changes before they can be applied on other platforms.

```
> grep " [0-9][0-9]*[A-Z0-9\-] * PRODCODE" * | wc -l
   341
```

The above `grep` command looks for all lines starting with a Cobol level number (cf. the `[0-9][0-9]`) where the data name declared in the line contains the string `PRODCODE`. We pipe the output of the `grep` command into the word count command `wc -l` to solely obtain the number of hits for `PRODCODE`.

Quantitative measure. For the programs that were supplied by our customer, there were 341 occurrences of `PRODCODE` in the `DATA DIVISION`s of the initial batch of programs. Apart from the programs, there were another 104 product codes in the supplied `COPY` books (including 54 in the *generated* ones). This simple query gave us a first indication of the volume of data declarations that needed to be changed.

Tooling in time. We will comment on technology for tooling later, but we want to state here very clearly that in this phase of the project one should either use very light weight tooling such as `grep`, or (advanced) tooling which is already *up and running*. One must avoid labour-intensive development of tooling in this phase because the problem analysis in architectural modification projects has usually to be completed before pricing on very short notice. This does not leave room for software development aimed at supplying tool support. We note that code exploration only has to be precise enough for cost estimation as opposed to the precision needed for the ultimate technical solution.

Sketch of the operational model.

Replace the data type `PIC 99`
in the declarations of data fields whose names contain `PRODCODE`
by `PIC 999`.

Effort estimation. The effort for implementing this requirement is pretty low. There are only 341 + 104 product code fields. The above (sketch of an) operational model is directly executable with even simple means of textual replacement. The required effort for implementing this model is less than 1 person day.[6] None of the project drivers for the PRODCODE project (cf. Section 2.4) seems to complicate the implementation of the requirement. For instance, syntax retention would be easily guaranteed because the required transformation operates on a very small focus.

4.3. Identification of undue assumptions

We will now discuss the steps to obtain a more detailed problem specification in the PRODCODE project. This involves reviewing the initial problem statement and the so-far gathered model. This review is meant to identify directions for deeper exploration.

The accuracy of the problem specification. Problems with simple requirements specifications are often underestimated, apparently many people stop after a first formulation of the operational model. Research by Boehm [6,7] shows that stopping at this moment is not a good idea. In the initial phase of a project, there is usually a factor of 16

[6] *1 person day* is meant to represent 8 h of work by a person representing the solution provider.

difference between optimistic and pessimistic estimates of the effort. Putting a little more effort into defining an operational model pays off considerably: when the requirements specification is known in detail, there is only a factor of three difference between high and low estimates. This is an improvement of over 80%. That is why our process for problem analysis is iterative.

Completing the naming conventions. In accordance with the findings of Boehm we had no reason to believe that we already converged to the final problem specification and we continued to investigate the problem in more detail. We decided to challenge the naming convention PRODCODE that was originally offered by the customer. To this end, we applied grep instructions as follows in order to find all declarations of fields with two digits:

```
> grep "PIC *99 | ... | wc -l"
    577
> grep "PIC *9(2) | ... | wc -l"
    186
```

The elisions indicate that these grep commands are slightly more elaborate to avoid matches with other data types whose prefix happens to match such as PIC 999 or PIC 9(2)V9(3). We browsed over the $577 + 186 = 763$ field declarations, and checked the use of some of them in the source code. We discussed a couple of candidates with the client, after which we added PRDCODE and PC to the naming conventions.

Accounting for disobeyed naming conventions. One can expect that sometimes the naming conventions are not obeyed. We used code exploration to determine if approved fields for product codes are moved to other fields whose name would not be covered by the naming conventions. For instance, the following query gives us all the lines with MOVE statements with a PRODCODE field as the source:

```
> grep "MOVE *[A-Z0-9\-]*PRODCODE" *
p4247v88:00228 MOVE PRODCODE IN LINK-FIELD-07 TO PRODCODE
p4247v88:00238 MOVE PRODCODE IN TAB-PROG(IND1) TO PRODCODE IN A-DAT.
p4737v13:00118 MOVE PRODCODE-BOOKING TO IND1.
p4737v13:00588 MOVE PRODCODE IN TAB-PROG(IND1) TO PRODCODE IN A-MAND.
...
```

One of the shown hits reveals the data name IND1 in line 00118 of program p4737v13, which obviously does not follow the prescribed naming conventions.

Recall—tooling in time. Note that we could have developed a data-flow analyser to accomplish the above task with more accuracy. We have seen people tempted to write the perfect data-flow analyser, control-flow analyser, slicer, dead-code detector etc. to solve such problems, only to find out that they did not need the tool, or that it took too much time. Therefore, we recall that, at this stage, it is better to use simple tools which do not require any up-front investment in tool development. Even if powerful infrastructure is available, the time for customisation to be applicable to the portfolio at hand might be non-affordable. For example, the advent of an in-house preprocessor, a so-far non-encountered cocktail of embedded languages, or simply a strange dialect is likely to trigger considerable customisation efforts for advanced technology—be it just one day, which is still too much. Besides customisation, the issue of explaining findings to the people at the client site should not be underestimated, which often speaks in favour of simple tooling.

The refined operational model. The above investigation of MOVE statements shows that many fields were used as product codes, while their names do not reveal their functionality. So, we need to improve our operational model to identify affected fields more precisely:

> *All field declarations that follow the naming conventions PRODCODE, PRDCODE, and PC are affected. In addition, all fields that occur in MOVE statements together with some other affected field are affected as well.*

We have used this style of specification also in Y2K and Euro-conversion projects, namely in the design of sophisticated tools for the identification of problem spots as needed for an impact analysis. The above model raises two issues with view on an estimation of effort:

- *Pollution*: We need to make sure that the identification of affected fields is precise, i.e., it does not suffer from false negatives or false positives.
- *Name resolution*: We need to be able to navigate from use sites of affected fields (in MOVE statements and elsewhere) to declaration sites.

We will now discuss these problems in more detail.

Pollution. A naïve algorithm for the identification of affected fields is likely to include too many fields. This can happen if some field is not just used for product codes but also for other types. Then, fields of all the other types might get included as well by transitive closure. To see if this problem was relevant, we investigated a few MOVE statements by looking at the types of the operands. Consider the following line of code which shows a data item TMP-ALPHA used in a MOVE statement together with a definite product code:

```
MOVE PRODCODE TO TMP-ALPHA.
```

The type of TMP-ALPHA turned out to be PIC X(80). This extra-large type and the further use of TMP-ALPHA in the program suggested that this field served as a heterogeneous buffer field. We must not qualify fields as affected just because they were used in a MOVE statement together with TMP-ALPHA.

Name resolution. While our initial operational model could be read as a kind of textual replacement command, the identification of affected fields via use sites complicates the situation. Namely, we start to interact with the language syntax and static semantics, i.e., we must be able to navigate from use sites such as MOVE statements to the corresponding declarations of the operands, which is not entirely trivial. That is, such name resolution potentially needs to handle compound references to data fields, which is illustrated with the following nested group field—notice the two occurrences of FLDX:[7]

```
00115   01      TOPREC.
00116    03     REC1.
00117     05    REC11.
00118       07  FLD111    PIC 99.
00119       07  FLDX      PIC XX.
00120     05    FLD12     PIC 99.
00121    03     REC2.
00122     05    FLD21     PIC X(10).
00123     05    FLDX      PIC 99.
```

[7] The grouping is visually emphasised by indentation as is common in Cobol, but the grouping is defined by the order of the level numbers.

There are some degrees of freedom for referring to components of such group fields. Here are legal references as they could possibly occur in MOVE statements and elsewhere:

- FLDX IN TOPREC IN REC1 IN REC11 refers to the first declaration.
- FLDX IN REC11 also refers to the first declaration.
- FLDX IN REC1 also refers to the first declaration.
- FLDX IN TOPREC IN REC2 refers to the second declaration.
- FLDX IN REC2 also refers to the second declaration.

In fact, the data name FLDX alone does not correspond to a valid reference as it is ambiguous. With regard to the PRODCODE project, we end up with the following question: Do affected fields require the use of compound references during name resolution? Fortunately, our further analysis revealed that such precision was not required.

Effort estimation. As precise name resolution is relatively expensive, we checked whether we would really need it in the PRODCODE project. To this end, we prototyped the identification of affected fields. We augmented this prototype such that inclusion of an overloaded name would be reported. It turned out that overloaded names did not occur except for fields that followed the naming conventions of product codes. So our analysis revealed that we could ignore compound names during name resolution.

The extra effort for a propagation of affected fields (as operationalised above) plus an ad hoc name resolution was estimated to account for 3 person days, adding to the earlier 1 person day for the initial operational model. We note that the effort has grown by a factor of four compared to the initial estimation, which is well in line with the findings by Boehm [6,7] regarding the precision of requirements specifications.

Type variations. Another component of the initial problem statement is also worth challenging. Our exploration of affected fields revealed that types other than PIC 99 were in use for product codes. We detected over ten different types. We will discuss the actual list later. For each type possibly a different expansion rule is necessary. For now, it is important to realise that the gained insight enables us to make a more realistic estimate regarding the increased effort for this project. This estimate is higher than the customer initially expected. Therefore it is a good idea to make sure the customer is made aware of the encountered complications. Among the types we found, other than PIC 99 there was one that puzzled us the most:

```
00083   01 SUBSCRIPTS.
00084      03 SUB-PC  PIC S9(4) COMP.
...
00294   MOVE PRODCODE TO SUB-PC.
```

The picture string S9(4) describes the type of a *signed* numeric data item with four digits. Why would one need a signed type? Why four digits? Note that we included a line of source code showing the connection between SUB-PC and PRODCODE. When we confronted the domain experts from the client site with the type S9(4), they could not offer any explanation. Such findings are a reality of modification projects, and such information can be used to build confidence in a thorough and fair cost estimation. Code exploration reveals

issues that come as a surprise to the domain experts but need to be understood in order to supply a solution for the modification problem.

4.4. Identification of usage patterns

Until now we have only considered declarations as possibly affected code locations. An important question to answer is what other usage patterns of product codes are present, whether any of these are affected as well, and whether they need adaptation. While we were identifying affected fields, we had already probed for `MOVE` statements to see if they revealed any additional affected fields. But now we want to systematically learn about *all* the usage patterns.

PRODCODE tracing. We developed a simple analysis to determine all occurrences of `PRODCODE`, the corresponding statement form and other operands used in the same context. To this end, we used `perl` for rapid tool development. The produced list looks as follows:

```
p4247v88,    749,   15,   4,   field,     IND2,    SET
p4247v88,    788,   39,   2,   numeric,     00,     IF
p4247v88,    795,   15,   4,   field,     IND1,    SET
p4737v13,    118,   46,   4,   field,     IND1,   MOVE
p4737v13,    318,   10,   4,   alpha,     '42',   MOVE
p4737v13,   1811,   39,   2,   numeric,     99,     IF
...
```

This list had 1477 entries corresponding to 1477 uses of product codes in the PRODCODE application. Each line of output consists of the program name, the line number, the columns, and the operand used in combination with a `PRODCODE` field. Also an indication of the kind of operand is shown: a `field`, a `numeric` literal, or an `alpha`numeric literal. Finally, every line of the report mentioned the statement form in which the `PRODCODE` field was found. The above report provides us with a useful overview in what contexts `PRODCODE`s are used. They are not just used in `MOVE` statements but also in `SET` conditions and `IF` statements. These findings trigger questions:

- The occurrences of literals ring a bell. Why is 99 used together with a `PRODCODE` field in an `IF` statement? This is an indication of hard-coded literals.
- What is the reason for using a `SET` statement for product codes? The `SET` statement seems to reveal that tables are subscripted by product codes.

These two patterns and all the others deserve a deeper analysis by an exhaustive case discrimination since they are potentially subject to adaptation. So we are clearly not yet done with a full problem analysis. The number of 1477 *uses* is a factor of five larger than the original number of merely affected declarations. This increase of effort is again fully aligned with Boehm's findings [6,7].

Hard-coded literals. Regarding the first question given above, we learned that the PRODCODE implementation heavily relied on the use of hard-coded literals for product codes. Most notably, such hard-coded literals occurred in `IF` and `MOVE` statements, where the boundary literals 99 and 100 were used in conditions to limit loops over legal product codes. As we will see later on, literals other than 99 and 100 were used elsewhere. Let us consider one occurrence of a literal in an `IF` statement together with a `PRODCODE` field:

```
01810   IF PRODCODE IN ENTRY1-PROG NOT NUMERIC
01811   OR PRODCODE IN ENTRY1-PROG > 99
01812      GO TO 5391.
01813   MOVE PRODCODE IN ENTRY1-PROG TO SUB-PROG.
```

While the initial problem statement assumed that there were 99 possible codes, the comparison ... > 99 reveals that codes greater than 99 also carry a meaning. Many affected fields were already of types like PIC 999 prior to data expansion such that they could store values greater than 99. Code snippets like the one above invited us to suspect usage of values greater than 99 as error codes. We also noticed that the implementation used dynamic type-checks to ensure that fields for product codes contain proper numeric values. That is, in the Boolean condition in line 01810, there is a check for NOT NUMERIC. This triggers new questions: Why would anybody check for a PRODCODE to be numeric? Were they not always numeric?

Revealing error codes. A simple way to look for potential error codes is to search for loops ranging over PRODCODEs. In the actual system, we could find many of those. Let us look at the following loop:

```
00758   5703.
00759       IF IND2 > 99
00760         GO TO 5705.
00761       PERFORM 5707.
00762       ADD 1 TO IND2.
00763       GO TO 5703.
```

Suppose that the analysis categorised IND2 as an affected field. The loop iterates over all product codes by incrementing the field IND2. For each product code, the paragraph 5707 is performed. The loop is terminated if the condition IND2 > 99 holds. Hence, 100 clearly is used as an error code. Our investigation did not reveal any alpha-numeric error codes. So we assume that the above test for the field to contain a numeric field is an indication of a defensive style of programming related to some irregular situations.

Adaptation of error codes. Apparently, fields that can hold the error value 100 must be of a type like PIC 999 already. Then, the question is if they need expansion to four digits or what other precautions are due. In fact, extension of fields of type PIC 999 is not needed unless we insisted on precisely 999 valid product codes. With 998 product codes or fewer, there will be still one code left to serve as error code. We state the following addendum to our operational model:

> *The literals 99 and 100 when occurring in simple comparisons involving an affected field as the second operand need to be replaced by the values newmax and newmax* $+1$. *Here, newmax denotes a parameter of the modification project, where newmax* < 999.

This formulation does not yet specify the affected patterns very precisely. However, the present formulation is considered sufficient for effort estimation.

Tables of product codes. In Cobol, there are five different sorts of statements that share the verb SET. The idiom which turned out to be used for product codes is the one to assign values to index fields for Cobol tables.

```
00235   01 TAB-PROG.
00236    03 ELEM01       OCCURS 99 INDEXED BY IND1.
00237     05 CODE-NOT    PIC X.
00238     05 MINIMUM     PIC 999V99.
00239     05 COMP-CODE   PIC 99.
...
00795    SET IND1 TO PRODCODE IN WG-MAND-NUM.
00796    MOVE 'Y' TO CODE-NOT IN ELEM01 (IND1).
...
```

In line 00236, the index field IND1 is declared and associated with the table ELEM01. In line 00795, a SET statement assigns a PRODCODE to IND1. In line 00796, a table access is performed where IND1 serves as an index field. So we have to conclude that the table ELEM01 is indexed by product codes.

Expansion of affected tables. After expanding the number of product codes, tables of product codes need to be expanded as well. When we encountered this new fact, we began to understand how a new upper limit for product codes could affect the performance of the system. The response time of the system and its memory requirements might change by a factor if we increase the size of tables from 100 to say 1000 entries. Of course this also depends on other factors like the size of the table entries, the complexity of the operations per entry, and the complexity of loops over such tables: e.g., are they linear or quadratic? We estimated that 299 product codes would be a good starting point, which could later on be revised if necessary. So we assumed *newmax* $= 299$. The operational model needs to be enhanced as follows to take findings about SET statements into account:

> *A SET statement reveals affected fields just as MOVE statements. The OCCURS clause for a table with an index field that is known to be affected has to be changed from* 99 *entries to newmax (*$= 299$*) entries.*

We can easily locate tables by recognising OCCURS clauses. A table is affected if it is INDEXED BY an affected field. Hence, we can easily change the OCCURS clauses to expand the table. So again, the operational model is straightforward to implement. A complication arises from the fact that the index of a table does not necessarily need to be declared explicitly, in which case the INDEXED BY clause is missing. Then, the link between index fields and the affected table has to be identified differently. In fact, this is again a simple form of name resolution. We estimated 2 person days effort for table expansion including a special-purpose name resolution.

4.5. Encountered subtleties

The code exploration revealed two issues that were particularly puzzling compared to things discussed so far. Firstly, numeric and alphanumeric types were freely used for product codes including hard-coded literals. We were initially unable to predict the implications of this irregularity. Secondly, there were a few tables that were not subscripted by product codes but still each table entry contained a product code. We were initially unable to decide on whether to expand or not. These subtleties will now be discussed in more detail, which underlines once more that architectural modifications tend to be more intricate than suggested by the deceptively simple problem statement that was communicated to us initially.

Suspicious tables. We are concerned with tables that involve a product code per entry. Such tables are potentially subject to expansion on the basis of the loose argument that if there are more product codes, then the table might also need to hold more entries, but in fact there are different idioms conceivable:

- *1–1 table*: A table is used to hold exactly one entry per valid product code. Hence, there must be as many table entries as product codes. This idiom, once revealed, requires table expansion.
- *Fixed table*: A table serves as a kind of buffer with a fixed number of entries. For instance, the size of the table might reflect the number of viewable entries on the screen. This idiom does not require expansion, or expansion might even be incorrect.
- *Open-ended table*: The number of entries in the table is bounded but it can be greater than the number of product codes. In this case, the new number of entries has to be defined per case.

Here is sample code for what could correspond to the first or the last idiom. The fragment fills a table in a loop reading *all* records from a file:

```
00337   01 WORK-PRODCODES.
00338    03 ELEM01           OCCURS 99.
00339     05 PRODCODE        PIC 99.
00340     05 SHORT-TXT       PIC X(8).
00341     05 VAL-MIN         PIC 999V99.
...
00934   0170.
00935       PERFORM TEST-PRODCODE.
00936       ADD 1 TO IND1.
00937       MOVE PRODCODE IN RECORD04 TO PRODCODE IN ELEM01(IND1).
...
00947       READ FILE04 NEXT RECORD
00948            AT END GO TO 0199.
00949       GO TO 0170.
00950   0199.
...
```

The field `IND1` is used as loop variable which is initially `ZERO`. The `OCCURS` clause for the table manifests the upper limit for the number of records in the file, namely 99. In this case, we would need to argue in favour of expansion because the structure of the loop suggests that records for all possible product codes could potentially be encountered. In fact, the loop by itself does not rule out several records per product code. Code exploration of the file-control entry for `FILE04` revealed that `PRODCODE` is declared to be a primary key of `FILE04`. Hence, there is at most one record per product code, and the normal increase from 99 to 299 table entries is appropriate.

One option is to adopt a defensive style of modification. That is, we expand all such tables regardless of further symptoms speaking in favour of expansion. Obviously, this approach is potentially wasteful. Also, customers often strictly dislike what they think are unnecessary modifications. Furthermore, there are even circumstances where an expansion can result in hazardous code. Most notably, expansion of a conceptually *fixed table* can lead to out-of-range problems on a screen or elsewhere. Fortunately, our code exploration revealed that there were only three suspicious tables in the PRODCODE project. This made us conclude that the issue of suspicious tables is less relevant in the

phase of cost estimation. We safely postponed dealing with the three suspicious tables until later.

Strange literals. Besides the maximum 99 and the error code 100, our simple code exploration also revealed several other hard-coded literals all over the system. In Section 4.4, we listed some usage patterns for product codes. We initially glanced over suspicious occurrences such as the alphanumeric literal `'42'` in an affected `MOVE` statement. What is the meaning of 42? Why are alphanumeric literals used here? We learned that codes other than 99 and 100 encoded distinguished product codes which had some special meaning in a certain program or at a certain time (e.g., in a migration program). The fact that some of these codes are alphanumeric is largely accidental. These hard-coded literals sometimes need to be changed (say, expanded) for subtle reasons as we will illustrate. Namely, consider an alphanumeric field for product codes:

```
00226   01    MAND-IN.
00227    03   PRODCODE PIC XX.
...
02491   MOVE '42' TO PRODCODE IN MAND-IN.
```

After expansion the `MOVE` statement will actually move the literal `'42'` to the field `PRODCODE IN MAND-IN`. Hence, the affected literal `'42'` has to be expanded to `'042'`. The PRODCODE application exercised basically all combinations of numeric and alphanumeric operands. By our systematic investigation, we also revealed that alphanumeric types like `PIC XX` were often used for product codes. This means that numeric and alphanumeric product codes coexisted in the program. So we would eventually need to make decisions for all such combinations. We confined ourselves to postpone working out details of literal expansion since we would only need to make a small number of decisions for expanding literals or not. Still the amount of literals and the subtle issue as such required adding 1 person day to the estimated effort.

4.6. Dissolved complications

Recall that we investigated whether or not a precise name resolution would be needed. Such investigations are crucial for limiting the costs of the project. There are several other complications that we dissolved in order to keep all analyses and transformations as simple as possible. Here is a list of dissolved complications:

- The seed-set construction might require liberalisation. For example, one could use conditions on the data declarations to qualify a field as element of the seed set. One could also use patterns of usage to support assumptions about elements of the seed set. Given the relatively reliable naming conventions in the PRODCODE project, this was not necessary.
- Sophisticated analyses can be used to uncover buffer fields or fields with several types of usage as opposed to pure PRODCODE fields. In the PRODCODE project, we restricted ourselves to simple heuristics, but we planned for human approval of affected fields. Given the size of the application, this was reasonable.
- Other syntactical forms can be taken into account, e.g., `MOVE CORRESPONDING` statements, arithmetic statements, file-access statements. In the PRODCODE project, we

concluded that such additional forms will not reveal more affected fields. For example, no arithmetic is performed on product codes.

- The analysis of affected fields can potentially be an inter-modular analysis. By this we mean that we track product codes across program boundaries by following all subprogram calls and all imports of COPY books for record descriptions and others. In the PRODCODE project, subprograms were not used for modularising application code, and naming conventions were obeyed in the COPY books. Hence, the identification of affected fields on a per-file basis was sufficient.
- The analysis of affected fields and the expansion of fields can require a precision where redefinitions (i.e., Cobol's form of unsafe variant records or unions) and MOVE statements with group fields as operands have to be taken into account. By means of code exploration we checked that this was not necessary.

There are a few more conceivable complications but we omit them here because they were not as relevant as those above. So in theory, all these issues have to be taken into account, but in practice we can neglect some of theme due to the specifics of the project. All these complications, when relevant, imply major extra effort, which we will illustrate by means of discussing the last item in the list given above.

Group MOVEs. Consider the following contrived code fragment:

```
01    PRODCODE        PIC 99.
01    WORK99.
 05   WORK99-FIRST    PIC X.
 05   WORK99-SECOND   PIC X.
...
MOVE PRODCODE TO WORK99.
```

That is, a PRODCODE field is moved to a group field WORK99 which happens to split up the product code into two characters. While this idiom was not exercised in the PRODCODE application, it is found in other projects that we carried out. For instance, date ingredients such as day, month, and year are often retrieved in this manner. The problem with this idiom is that the group field will also count as an affected field, which necessitates a generalised operational model for expansion. Also, the use of the positions WORK99-FIRST and WORK99-SECOND in other statements could be relevant for the identification of further affected fields. Furthermore, some code might rest on the assumption that there are only two positions. Such code needs an update as well. Without the finding that these problems were irrelevant for the PRODCODE project we would have embarked on more complex algorithms such as those in [24,72].

Redefinitions. Identification of affected fields and their expansion becomes further complicated if product codes occurred in redefinitions. This is Cobol's facility to provide multiple interpretations for stored data. We illustrate redefinitions by overlaying the two fields that were stored separately above:

```
01    PRODCODE        PIC 99.
01    PRODCODE-XX     REDEFINES PRODCODE.
 05   WORK99-FIRST    PIC X.
 05   WORK99-SECOND   PIC X.
```

Hence, we could access product codes in two ways, either as a two-digit value or position-wise. Again, this is a contrived fragment. Fortunately, the many redefinitions

in the PRODCODE application did not involve product codes. Other languages also offer related language constructs, e.g., unions in C. The use of such constructs is known to considerably complicate reasoning about programs. Redefinitions are pervasively used in Cobol programs. Especially in older software, redefinitions are used to save space by imposing two different interpretations on a block in memory or a file record. In such cases, expansion of both the primary and the overlaid data structure would not be justified. However, it is hard to determine if a redefinition is meant to provide an alternative view on the same data vs. an unrelated data structure.

4.7. Convergence of analysis

A useful idea to feed the problem specification process is to pick up one program from the code base and to perform some obvious changes manually. This approach will usually result in an intensive exploration of the particular program, and it will trigger the implementation of simple queries for code exploration. These queries are applicable to the full code base. The process will reveal many aspects of the solution strategy. In analysing code patterns, and implementing exploratory analyses, one can apply a worst-case selection criterion. For instance, the very first program should be one of the more complex programs, and one should cover simple and more complex cases. At that point, more code exploration will not necessarily give a more accurate cost estimation. There is no formula to detect the required amount of code exploration, but a form of time boxing is a good idea here: try to find as much as possible prominent aspects of the problem specification, and after that inventory make a new effort estimation.

In the PRODCODE project, the result of the problem specification phase was a collection of examples, scenarios, reports, and raw solution options—as illustrated above. This kind of problem specification was to a large extent comprehensible to the system owners. Thereby, a solid basis for a cost estimation was formed. In addition we came up with a management summary of how many changes in which context were probably necessary. This summary appears as follows:

```
Filename,  FS,  WS,  CY,  XX,  99,  MV,  ST,  IF,  MISC
p4247v88,   0,   3,   9,   2,   1,  21,  13,  16,   2
p4737v13,   0,   0,   5,   0,   0,   1,   1,   1,   0
p7766v13,   0,   9,  18,   0,   0,  22,   0,   7,  21
p7887v08,   0,   9,   6,   0,   0,  10,   0,   1,   1
p7888v15,   0,   1,   4,   0,   0,   4,   0,   4,   0
p7889v22,   1,   5,   5,   0,   0,   7,   0,   5,   4
...
```

The report lists all programs and the corresponding counts of `PRODCODE` for the respective pattern. We used the following abbreviations for the context of the patterns:

FS	affected code in file section
WS	affected code in working storage section
CY	affected code in `COPY` books
XX	affected code in alphanumeric literal context
99	affected code in numeric literal context

MV	affected code in `MOVE` statements
ST	affected code in `SET` statements
IF	affected code in `IF` statements
MISC	affected code in other statements.

All in all, there were 1818 code patterns that directly involved `PRODCODE`. This number is five times larger than the original estimate of 341 `PRODCODE` declarations.

5. Project economics

Now that we know the impact of the problem, we can deal with the pricing issue. An important characteristic of this type of project is that customers tend to demand a fixed price, for fixing a problem that occurs only once. The fixed price often origins from the handcraft-directed thinking of the customer: they initially envision a black box tool to do the job, which has a price, a fixed price. Being in that position, it is mandatory to come up with an accurate cost estimation. Also, evidence regarding the specifics and the complexity of the problem must be gathered. Thereby, the customer can be convinced that the estimation is reasonable.

5.1. Cost estimation and contract signature

The problem analysis, as described in Section 4, was completed in about a week. To this end, the analysis phase was covered by a consultancy contract for three person days with the problem specification as deliverable. While we were taking the lead in working out the problem specification, we also arranged for daily meetings with the client, complemented by phone calls and email threads. This guaranteed the necessary customer involvement. The idea of separate contracts for analysis and the rest of the project is that the customer stays in power, i.e., the customer can decide whether to continue the project or not, while the deliverable of the analysis phase, i.e., the problem specification, becomes the customer's property. This specification is supposed to be of use for the system owner regardless of the continuation of the project.

To initiate the continuation of the project, we submitted an effort distribution and cost estimation in the form of a draft contract. This submission offered a fixed price, which was calculated based on an effort estimation for 15 person days. To this end, the following activity-based effort distribution was listed:

- 3 working days for further code exploration and meetings with the customer: the expected results are the detailed modification rules, and detailed side conditions for service delivery, e.g., the kind of documentation.
- 3 working days for tool development for analysis of affected fields.
- 1 working day for tool development for picture-string expansion.
- 2 working days for tool development for table expansion.
- 2 working days for tool development for literal expansion.
- 2 working days for manual adaptations of remaining affected patterns, and recording them in transformation scripts.
- 1 working day for documentation of modifications.

Problem	Count
Classes of changes	
Picture expansion	275
Table expansion	70
Maximum expansion	126
Alphanumeric literal expansion	14
Numeric literal expansion	48
Adaptation of DISPLAY	11
FILLER contraction	12
Other adaptations	41
All changes	597

Fig. 3. Management summary of the PRODCODE project.

- 1 working day for running the mass-conversion on the production code, and sanity checking the mass-conversion results.

This effort distribution was defended during a meeting prior to contract signature. There was enough information available to make a good estimate. So we could agree on the terms, and signed a fair contract. This contract also fixed a few side conditions. Most notably our liability was limited by the fact that the modification of 4GL code and others would not be carried out by us. Also, the responsibility for testing was assigned to the customer. Furthermore, the contract stated that detailed documentation of all the performed changes had to be delivered.

5.2. Management summary

On completion of the project, the customer was provided with a kind of a management summary of the performed changes. This summary helped to convince the client that the agreed price was reasonable. An overview of numbers of changes and categories is shown in Fig. 3. Note that the compilation of such an overview was straightforward in the view of the requirement to document all changes. We recall that at one moment in time we found 341 declarations and 1477 use sites of `PRODCODE`s. Of course, not all the locations where affected code is found, needed an update. From the summary we can see that 597 actual changes were made. All the affected spots were detected by our tools for code exploration. Most of the changes were covered by generic transformation rules. The remaining changes were performed manually, but they were recorded in patch scripts so that we can replay them.

5.3. The cost and risk dimensions

We want to briefly argue that an architectural modification effort using automated transformations was the right way of addressing the PRODCODE problem. To this end, we summarise the costs and the risks of a manual approach. We will omit a discussion of

risks of automated modifications or, more generally, re-engineering, but we refer to Sneed's work [84] to this end.

Costs of a manual approach. From benchmarking data it is known that maintenance programmers can on the average deal with 8 bugs a month [35]. Other data indicates that 47% of the time a maintenance programmer is occupied with *understanding* the problem, in order to make an actual change [63]. From the management summary provided in Fig. 3 we can see that 597 changes had to be made. They were grouped in seven named categories, and additionally 41 changes did not fit any of the main categories. So there were $41 + 7 = 48$ *types* of change. By lack of further information we assume that applying a type of change is similar to fixing a bug, hence we can apply 8 types of changes a month, 48 changes in 6 months. Note, that applying the 47% of [63] gives us about 3 months for understanding alone. We made changes, while the client tested the changed code, and carried out complementary changes of the DB2 tables and others. This implied an additional 10 working days at the client site. Total effort was 28 working days, while the benchmarked estimate was 6 months, which is about 120 working days. So our automated process to architectural modification resulted in an improvement of a factor of four when compared to a conservative estimation for tackling the problem in the course of regular maintenance.

Problems with code volume as a metric. Some people might think that 100 working days for such a project by hand is too much. One way of looking at it is that this project contained about 100 files. So that is one day per file for a relatively small change. However, code volume is not a useful metric. Ted Keller showed this with an interesting analogy. Suppose that for gall bladder removal, the average duration is 45 min and the costs are $2,000. Suppose that for the average pea-sized brain tumour 8–10 h time at a cost of $10,000 is necessary. If a surgeon's effort is merely measured by tissue volume, one will obtain nonintuitive results. Suppose that a gall bladder is 20 times larger than a pea-sized brain tumour. Then neuro-surgeons are 100 times more expensive (per tissue volume), are 260 times less productive (per tissue volume), and general surgeons should be able to remove about 26 pea-sized brain tumours per hour for $103 each [40]. Keller used this to show that looking at code volume in maintenance projects leads to nonintuitive results. So, 100 working days for the PRODCODE problem is not much for such an effort, if one succeeds at all with the manual approach.

Risks of a manual approach. Apart from the *cost dimension* there is a *risk dimension* as well. In fact, this dimension is very important, and must not be missed by looking at the cost issue only. Suppose this type of mass-change project is done by hand. Then in many cases, the team loses oversight pretty quickly: which solutions are applied in which case, which programs are already updated, and which not. Which order to apply for which type of change, and if there is an order, how to keep track of this process. Namely, the conceptual scope of the average maintenance programmer is the program, and not the entire system. So when a change is made to a program with the goal to modify the structural integrity of the system, the program scope is not sufficient. Manual modification projects often fail, or are avoided in the first place, leading to years of patchwork incrementalism. If these mass-change projects are avoided, then this is indeed either because of the cost dimension

or of the risk dimension. It is our experience that *automated* modification efforts perform much better in both dimensions.

An example of failure. Let us give an example to demonstrate the risks of a manual approach. The example concerns the preparation of a system for a Euro conversion. Several manual efforts to do this modification effort had failed. At this stage, one of our colleagues was hired and resolved the problem by means of a 400 hour managed and automated modification project. The reason for earlier failure of the manual attempts was that not only a lot of changes needed to be made, but also, the changes were dealing with seemingly random alphanumeric identifiers with an average length of 20 characters. These systematic identifiers had to be recognised, and partly changed. Data structures needed to be phased out to `COPY` books, using the macro function called `REPLACED BY` where parts of the alphanumeric identifiers were factored out and put in the parameters of the replacement macro. All these identifiers looked alike, but were not exactly the same, so that typographical errors could hardly be detected. This status prevented any human being from implementing large-scale changes to this scheme since it was completely ingrained in the entire system.

So our calculation on productivity improvement has to be augmented by a risk dimension: our calculation shows how much one saves in case someone happens not to fail to do it by hand.

6. Design of the solution

The phase following the problem analysis deals with the detailed design of the solution. That is, we will now work out rules for the analysis and transformation of the PRODCODE application. We will describe the rules by means of examples, and some rules will be subjected to a formal specification. The rules are given in sufficient detail so that they can serve as the input for actual tool implementation.

The specification formalism. Let us briefly explain the specification formalism that we use. Our algebraic specification consists of equations of the form

$$\frac{s_1 = t_1, \ldots, s_n = t_n}{s = t}$$

where $s, s_1, \ldots, s_n, t, t_1, \ldots, t_n$ are (open) terms over a given signature. Such an equation is interpreted as a rewrite rule from left to right; given a ground term which matches with the left-hand side of the equation (the open term s), such that all conditions are fulfilled (all conditions $s_i = t_i$ evaluate to true), then s is rewritten to t. The variables that occur in the terms are bound to closed terms by matching. The specifications given in this section were executed by the ASF + SDF Meta-Environment—a system which implements conditional term rewriting with concrete syntax [44]. Our specification relies on a special idiom supported by ASF+SDF, namely generic traversal functionality [11,12]. Function symbols can be marked to model traversal of their first arguments, e.g., bottom-up or top-down traversal. The remaining arguments are parameters of the traversal. As a

$$\frac{\begin{array}{rcl}\text{Seed}(Prog_1, \emptyset) & = & Seed, \\ \text{Propagate}(Prog_1, Seed, \text{MaxIterations}) & = & Affected, \\ \text{PicExpansion}(Prog_1, Affected) & = & Prog_2, \\ \text{MaxExpansion}(Prog_2, Affected) & = & Prog_3, \\ \text{LiteralExpansion}(Prog_3, Affected) & = & Prog_4, \\ \text{TableExpansion}(Prog_4, Affected, Prog_1) & = & Prog_5\end{array}}{\text{Main}(Prog_1, \text{MaxIterations}) = Prog_5} \quad (1)$$

Fig. 4. Top-level specification.

result, specifications become radically more concise since only constructs of interest need to be covered by equations.

6.1. The top-level specification

In Fig. 4, the top-level specification of the PRODCODE modification is given by a single conditional equation. It expresses that the whole modification consists of the following steps:

- In the first condition, a seed set is determined by the function Seed for the given program $Prog_1$. This is a set of fields that match certain naming conventions. The seed set is the starting point for determining affected fields.
- In the second condition, the function Propagate extends the seed set with all other affected fields. This will be done by analysing the statements in the source code of $Prog_1$, based on a maximum number of iterations.
- In the third condition, the function PicExpansion expands picture strings of affected field declarations $Prog_1$. This will be an elaboration of the simple case to turn `PIC 99` into `PIC 999`. This results in an intermediate result $Prog_2$.
- In the fourth condition, the function MaxExpansion expands all literals that refer to the maximum number of product codes. This concerns the literals 99 and 100 in `IF` statements and elsewhere. This transformation produces yet another intermediate result $Prog_3$.
- In the fifth condition, the function LiteralExpansion expands all literals that refer to product codes. This is meant to compensate for the mixture of numeric and alphanumeric product codes. This transformation produces yet another intermediate result $Prog_4$.
- In the sixth condition, the function TableExpansion expands all tables that are subscripted by product codes. The transformation operates on $Prog_4$ and it computes the final result $Prog_5$. However, the function TableExpansion takes the original program $Prog_1$ as an additional parameter for name resolution.

These steps are discussed one by one in the following. In the last part of the section, we will discuss some specific changes separately, and we will explain the approach to documenting all the required changes.

$$\frac{\text{IsProdcodeName}(Id) \text{ and } \text{IsProdcodeLength}(Str) = \text{true}}{\text{Seed}(\textit{Level Id Dd-items}_1 \text{ PIC } \textit{Str Dd-items}_2, \textit{Seed}) = \textit{Seed} \cup \{Id\}} \quad (2)$$

$$\text{Seed}(\textit{Level Id Dd-items}, \textit{Seed}) = \textit{Seed} \quad \textbf{otherwise} \quad (3)$$

$$\text{IsProdcodeName}(\texttt{*PRODCODE*}) = \text{true} \quad (4)$$

$$\text{IsProdcodeName}(\texttt{*PRDCODE*}) = \text{true} \quad (5)$$

$$\text{IsProdcodeName}(\texttt{*PC*}) = \text{true} \quad (6)$$

$$\text{IsProdcodeName}(Id) = \text{false} \quad \textbf{otherwise} \quad (7)$$

$$\text{IsProdcodeLength}(Str) = (2 \leq \text{length}(Str) \text{ and } \text{length}(Str) \leq 4) \quad (8)$$

Fig. 5. Identification of the seed set—specification.

6.2. Identification of affected fields

We first identify the *seed set*, which contains the affected fields that are recognisable in the code based on naming conventions. Then, a *propagation* step needs to be performed repeatedly by computing the closure of fields with the same type of usage.

Finding the seed set. During the problem specification phase we have checked the naming conventions that were initially provided by the customer and we have found some additional ones. As a result, we knew that the names `PRODCODE`, `PRDCODE`, and `PC` are used for product codes. These names can also occur as prefixes or postfixes of other data names. In Fig. 5, this is specified accordingly. We define the function Seed, which accumulates a seed set denoted by *Seed*. In the top-level specification given in Fig. 4, the function Seed was applied to the complete program $Prog_1$, whereas the only location where a match is possible at all, is in Cobol's `DATA DIVISION`. As we rely on generic traversal functionality, we do not need to specify all the trivial rewrite rules to navigate to the relevant patterns. We explain the remaining equations of Fig. 5. The function IsProdcodeName is used to check whether an identifier matches with any of the search patterns. To this end, the equations employ a wild-card notation; cf. "$*$". Furthermore, we check the length of the fields with the function IsProdcodeLength. During code exploration we encountered that most affected fields have length two, but we also encountered product code fields of length 3 or 4. While fields of length 3 or 4 may not have to be expanded, they still could be needed to reveal affected fields by propagation. It is strongly recommended to check the seed set for false positives, as they may pollute the remaining steps in the analysis.

Propagation. The purpose of propagation is to identify the data fields in a program that are of the same type of usage [22] as the fields in the seed set. We have subdivided propagation into two phases. Firstly, there is a one-step algorithm that detects affected fields in statement forms. Secondly, there is an iteration with a suitable stop condition. In the PRODCODE project, this straightforward propagation algorithm did the job.

$$\text{OneStep}(\text{MOVE } Id_1 \text{ TO } Id_2, \mathit{Affected}) = \text{OneStepUnion}(\{Id_1, Id_2\}, \mathit{Affected}) \tag{9}$$

$$\text{OneStep}(\text{SET } Id_1 \text{ TO } Id_2, \mathit{Affected}) = \text{OneStepUnion}(\{Id_1, Id_2\}, \mathit{Affected}) \tag{10}$$

$$\frac{\mathit{Fields} \cap \mathit{Affected} \neq \emptyset}{\text{OneStepUnion}(\mathit{Fields}, \mathit{Affected}) = \mathit{Fields} \cup \mathit{Affected}} \tag{11}$$

$$\text{OneStepUnion}(\mathit{Fields}, \mathit{Affected}) = \mathit{Affected} \quad \textbf{otherwise} \tag{12}$$

Fig. 6. One-step propagation—specification.

$$\text{Propagate}(\mathit{Prog}, \mathit{Affected}, 0) = \mathit{Affected} \tag{13}$$

$$\frac{\text{OneStep}(\mathit{Prog}, \mathit{Affected}) = \mathit{Affected}}{\text{Propagate}(\mathit{Prog}, \mathit{Affected}, n+1) = \mathit{Affected}} \tag{14}$$

$$\frac{\text{Propagate}(\mathit{Prog}, \mathit{Affected}_0) = \mathit{Affected}_1}{\text{Propagate}(\mathit{Prog}, \mathit{Affected}_0, n+1) = \text{Propagate}(\mathit{Prog}, \mathit{Affected}_1, n)} \quad \textbf{otherwise} \tag{15}$$

Fig. 7. Iterated propagation—specification.

One-step propagation. Fields used in `MOVE` and `SET` statements have the same type of usage, and thus, the constructs provide input for one-step propagation. Consider the following `MOVE` statement:

```
MOVE PRODCODE TO PC IN CUSTOMER-RECORD.
```

In this example, one-step propagation does not reveal any new affected fields because both `PRODCODE` and `PC` are already covered by the naming conventions for product codes. In the following `SET` statement, the field `IND1` is uncovered by one-step propagation:

```
SET IND1 TO PC IN CUSTOMER-RECORD.
```

One-step propagation is formalised in Fig. 6. In a `MOVE` statement and a `SET` statement, the two involved fields have the same type of usage; see equations (9) and (10) in Fig. 6. These equations also suggest how one-step propagation can be readily arranged for other syntactical patterns. If a set *Fields* of fields of the same type of usage overlap with the set *Affected* of affected fields so far, then we add all *Fields* to *Affected*; see Eq. (11) in Fig. 6. Otherwise, the set *Affected* is preserved; see Eq. (12) in Fig. 6.

Iterated propagation. The definition of propagation analysis is completed by iterating one-step propagation until no new fields are found, or until the maximum iteration count is reached. The top-level Eq. (1) in Fig. 4 stated that the set of affected fields is initialised with the seed set, and the parameter position for an iteration count is initialised with the maximum count. The iteration of the propagation is defined in Fig. 7. Eq. (13) in Fig. 7

simply states that the iteration stops whenever the iteration count has reached zero. In the other two equations, the function for one-step propagation is applied. Eq. (14) in Fig. 7 states that the iteration stops if no new affected fields are found, whereas Eq. (15) in Fig. 7 states that we continue the iteration otherwise.

Pollution problems. As we have discussed earlier, the computation of affected fields has to be done with care. In the PRODCODE project, we did indeed not include fields with king size formats such as buffers during propagation. For brevity, this is not reflected in the above specifications. In other projects, we have found it useful to employ negative name patterns and other criteria to avoid false positives. In the PRODCODE project, iterated propagation converged already after the first step, that is, the second step did not reveal additional affected fields. In our experience, this is usually a good indication for absence of pollution problems.

6.3. Picture-string expansion

We start now with picture-string expansion. Our intuition is of course to expand the type `PIC 99` to `PIC 999`, but we will have to discuss several other types as well. Let us first illustrate expansion for a `DATA DIVISION` fragment of a program:

```
00115   01      STRA-FIELDS.
00116    03     STRA-MAND.
00117     05    STRA-DATA.
00118       07  PC              PIC 99.
00119       07  FUN             PIC 9(4).
00120     05    SKI              PIC 999.
```

In line 00118 we see a `PC` field at level `07` whose name matches with conventions for product codes. The field must be expanded since the picture string is indeed too short. The expansion leads to the following code:

```
00115   01      STRA-FIELDS.
00116    03     STRA-MAND.
00117     05    STRA-DATA.
00118       07  PC              PIC 999.
00119       07  FUN             PIC 9(4).
00120     05    SKI              PIC 999.
```

We underlined the expanded picture string. Several other types are used for product codes as well, e.g., `PIC 999` (i.e., three digits), `PIC XX` (i.e., two alphanumeric positions), and `PIC BXX` (i.e., three alphanumeric positions with a blank if the first position happens to be zero). In Fig. 8, we map all data types for product codes, that were present in the PRODCODE application, to expanded types. Affected fields of other types, e.g., of type `PIC 999`, are preserved. Here, we assume that these other types are capable of holding at least three digits.

In Fig. 9, the specification for picture-string expansion is given. Eq. (16) in Fig. 9 deals with the pattern of so-called data description entries, i.e., field declarations. The first condition checks if the data name *Id* at hand is in the set of affected fields. The second condition performs the actual picture-string expansion (if any)—as it was specified in Fig. 8. The first argument of the function PicExpansion is an arbitrary field declaration. Using associative list pattern matching its picture string Str_0 is obtained and turned into

$$
\begin{array}{lcl}
\text{MapString}(\texttt{99}) & = & \texttt{999} \\
\text{MapString}(\texttt{9(2)}) & = & \texttt{9(3)} \\
\text{MapString}(\texttt{BB9(2)}) & = & \texttt{B9(3)} \\
\text{MapString}(\texttt{BB9(02)}) & = & \texttt{B9(03)} \\
\text{MapString}(\texttt{BBBB9(2)}) & = & \texttt{BBB9(3)} \\
\text{MapString}(\texttt{B(8)9(02)}) & = & \texttt{B(7)9(03)} \\
\text{MapString}(\texttt{S9(2)V}) & = & \texttt{S9(3)V} \\
\text{MapString}(\texttt{S9(02)V}) & = & \texttt{S9(03)V} \\
\text{MapString}(\texttt{XX}) & = & \texttt{XXX} \\
\text{MapString}(\texttt{X(02)}) & = & \texttt{X(03)} \\
\text{MapString}(\texttt{X(002)}) & = & \texttt{X(003)} \\
\text{MapString}(\texttt{BBXX}) & = & \texttt{BXXX} \\
\text{MapString}(\mathit{Str}) & = & \mathit{Str} \quad \textbf{otherwise}
\end{array}
$$

Fig. 8. Picture-string expansion—specification (part I).

$$
\frac{\begin{array}{c} \mathit{Id} \in \mathit{Affected} \;=\; \text{true}, \\ \text{MapString}(\mathit{Str}_0) \;=\; \mathit{Str}_1 \end{array}}{\begin{array}{c} \text{PicExpansion}(\mathit{Level\ Id\ Dd\text{-}items}_1\ \text{PIC}\ \mathit{Str}_0\ \mathit{Dd\text{-}items}_2., \mathit{Affected}) = \\ \mathit{Level\ Id\ Dd\text{-}items}_1\ \text{PIC}\ \mathit{Str}_1\ \mathit{Dd\text{-}items}_2. \end{array}} \tag{16}
$$

Fig. 9. Picture-string expansion—specification (part II).

its expanded equivalent Str_1. Note that this specification will be more complicated in case name resolution takes compound data reference into account.

6.4. Maximum expansion

As pointed out earlier, literals need to be adapted if they exercise boundary values of product codes. Code exploration revealed that the boundary values 99 and 100 were explored in conditions of `IF` statements, only. The required adaptations were of the following form.

```
PC > 99        ⇒   PC > 299
PC NOT > 99    ⇒   PC NOT > 299
PC = 100       ⇒   PC = 300
PC NOT = 100   ⇒   PC NOT = 300
```

Here `PC` denotes the name of an affected field. We used 299 as the new maximum. Flipped variations on the patterns are omitted for brevity. Maximum expansion is illustrated with the following code fragment:

$$\frac{Id \in Affected \;=\; \text{true}, \quad \text{MapBoundary}(Num_0) \;=\; Num_1}{\text{MaxExpansion}(Num_0\ Rel\ Id) = Num_1\ Rel\ Id} \tag{17}$$

$$\frac{Id \in Affected \;=\; \text{true}, \quad \text{MapBoundary}(Num_0) \;=\; Num_1}{\text{MaxExpansion}(Id\ Rel\ Num_0) = Id\ Rel\ Num_1} \tag{18}$$

$$\text{MapBoundary}(99) = 299 \tag{19}$$

$$\text{MapBoundary}(100) = 300 \tag{20}$$

$$\text{MapBoundary}(Num) = Num \tag{21}$$

Fig. 10. Expansion of table boundaries.

```
00758   5703.
00759        IF IND2 > 99
00760          GO TO 5705.
00761        PERFORM 5707.
00762        ADD 1 TO IND2.
00763        GO TO 5703.
```

Propagation revealed that `IND2` is an affected field in the program. The condition in line `00759` refers to the number of product codes, and hence it needs to be adapted accordingly:

```
00758   5703.
00759        IF IND2 > 299
00760          GO TO 5705.
00761        PERFORM 5707.
00762        ADD 1 TO IND2.
00763        GO TO 5703.
```

In principle, patterns other than those discussed above are conceivable because comparisons can be formed in various ways. It is a good idea to restrict transformation rules to actual code patterns present in the given code base. As an aside, thinking of normalisations is not always a good idea. Firstly, a normaliser can require substantial work. Secondly, normalisation can make it more difficult to meet side conditions like syntax retention.

In Fig. 10, maximum expansion is specified. The rules cover the special patterns that we listed above. The transformation function MaxExpansion is applied to conditional expressions consisting of the relational operator *Rel* and two operands. The variable *Rel* matches with relational operator symbols like "=", ">", and "<" but also with their negated counterparts like "NOT =". There are two Eqs., (17) and (18) in Fig. 10, since an affected field can serve as a left or a right operand, while the other operand is then required to be a numeric literal. The last three Eqs., (19)–(21) in Fig. 10, specify the actual expansion of literals.

6.5. Literal expansion

When we analysed the problem, we learned that both numeric and alphanumeric types were used as product codes. Furthermore, code exploration revealed that both numeric and alphanumeric literals were hard coded in many cases. In particular, literals occurred in patterns like this:

- `MOVE '42' TO PC.`
- `MOVE 42 TO PC.`

Here, `PC` is an affected field of whatever type. Such patterns raise the question whether literals need to be expanded when the corresponding fields are expanded. The semantics of `MOVE`s for numeric and alphanumeric types is well defined, but a bit subtle to interpret. Virtually every combination of numeric or alphanumeric source and target for any length of picture string is valid, and the result is regulated by a number of rules. The following table explores some combinations. When a literal `'42'` or `42` is moved to a field `PC`, the actual result depends on the picture string. In the following table, we denote a space by the symbol "␣".

		PC PIC XXX	PC PIC 999
alpha-numeric literal	`MOVE '42' TO PC`	␣42	42␣
numeric literal	`MOVE 42 TO PC`	␣42	042

We obtained this knowledge by writing a tiny test program because this seemed to be the most reliable approach to us. Even experienced programmers cannot predict these cases very reliably. From this table we concluded that literals like `'42'` or `42` should be expanded as well, to respectively `'042'` and `042`.

Patterns involving literals. It is clear that literals can occur in a number of syntactical patterns, not just in `MOVE` statements. By a systematic code exploration, we narrowed down all possible patterns to those that were actually exercised in the PRODCODE application:

- source operands of `MOVE` statements;
- in conditions of `IF` statements;
- in declarations of condition names.

We extend the operational model accordingly:

> *If a literal, alpha-numeric or numeric, of length 2 is assigned to or compared with a product code field, a leading zero should be added. Literals can also occur as part of the declaration of condition names for product codes. These literals need to be expanded in the same manner.*

The following fragment illustrates literal expansion in `IF` statements:[8]

[8] This fragment uses Cobol's so-called abbreviated combined relation conditions. That is, the condition in the `IF` clause is abbreviated in a manner assuming that the left operand of the first comparison and the comparison operator "=" are inherited to the subsequent abbreviated comparisons. For instance, `A = '1' OR '2'` abbreviates `A = '1' OR A = '2'`.

$$\frac{Id \in Affected \;=\; \text{true}, \quad \text{MapLiteral}(Lit_0) \;=\; Lit_1}{\text{LiteralExpansion}(\text{MOVE } Lit_0 \text{ TO } Id, Affected) = \text{MOVE } Lit_1 \text{ TO } Id} \tag{22}$$

$$\frac{Id \in Affected \;=\; \text{true}, \quad \text{MapLiteral}(Lit_0) \;=\; Lit_1}{\text{LiteralExpansion}(Id\ Rel\ Lit_0, Affected) = Id\ Rel\ Lit_1} \tag{23}$$

$$\frac{Id \in Affected \;=\; \text{true}, \quad \text{MapLiteral}(Lit_0) \;=\; Lit_1}{\text{LiteralExpansion}(Lit_0\ Rel\ Id, Affected) = Lit_1\ Rel\ Id} \tag{24}$$

$$\text{MapLiteral}(\text{"'"}\ c_1\ c_2\ \text{"'"}) = \text{"'"}\ \text{"0"}\ c_1\ c_2\ \text{"'"} \tag{25}$$

$$\text{MapLiteral}(c_1\ c_2) = 0\ c_1\ c_2 \tag{26}$$

$$\text{MapLiteral}(Lit) = Lit \quad \textbf{otherwise} \tag{27}$$

Fig. 11. Literal expansion—specification.

```
01731   IF PRODCODE IN WG-MAND-ALPHA = '01' OR '02' OR '13'
01732      NEXT SENTENCE
01733   ELSE
01734      GO TO 0377.
```

We need to expand all the literals in the compound condition, since they are are compared with the field `PRODCODE IN WG-MAND-ALPHA`:

```
01731   IF PRODCODE IN WG-MAND-ALPHA = '001' OR '002' OR '013'
01732      NEXT SENTENCE
01733   ELSE
01734      GO TO 0377.
```

Several of the programs, in which hard-coded literals occurred, were eventually suspected to be dead. In a few cases, a maintenance programmer argued that the programs used literals for product codes because they implemented a one-shot data conversion at some point in time. Here is a code fragment from such a suspected conversion program. This fragment also illustrates that literals for product codes can occur as part of data-field declarations:

```
00159   03 COND-OCCUPIED PIC 99 VALUE ZERO.
00160       88 IS-OCCUPIED-CODE
00161          VALUES 07, 15, 16, 31, 37,
00162                 84, 85, 86, 90, 98.
```

In line 00160, the condition name `IS-OCCUPIED-CODE` is declared. This name can serve as a form of condition, which is true whenever the field `COND-OCCUPIED` holds one of the `VALUES 07, 15, ..., 90, 98`. We expanded all the literals.

Literal expansion is specified in Fig. 11. Eq. (22) in Fig. 11 covers literals in MOVE statements. Eqs. (23) and (24) in Fig. 11 cover literals in comparisons. For brevity, we omit some rules that are needed to deal with more complex forms of conditions, e.g., abbreviated combined relation conditions. The actual expansion of literals is defined in Eqs. (25)–(27) in Fig. 11. Literals of length two are taken apart, and a padding zero is added.

6.6. Table expansion

When a data field is subscripted by an affected field, then the table that hosts the subscripted field has to be expanded. Such a table normally holds 99 entries before expansion, and *newmax* entries after expansion. For illustrative purposes, we continue with *newmax* = 299. The following fragment involves a table declaration, some table accesses, and the code which uncovered the corresponding index field to be a product code during propagation:

```
00235   01   TAB-PROG.
00236        03   ELEM01          OCCURS 99 INDEXED BY IND1.
00237             05  MINIMUM     PIC 999V99.
00238             05  COMP-CODE   PIC 99.
...
00589        MOVE MINIMUM IN LINK-P3233-P TO MINIMUM
00590          IN TAB-PROG (IND1).
00591        MOVE COMP-CODE IN LINK-P3233-P TO COMP-CODE
00592          IN TAB-PROG (IND1).
...
00787        SET IND1 TO PRODCODE IN WG-MAND-NUM.
```

Only in the last line, many lines away from the table access, we see from the SET statement that IND1 is connected with a PRODCODE. The transformed part of the above fragment appears as follows:

```
00235   01   TAB-PROG.
00236        03   ELEM01          OCCURS 299 INDEXED BY IND1.
00237             05  MINIMUM     PIC  999V99.
00238             05  COMP-CODE   PIC  99.
```

We omit the rewrite rules for table expansion, but we will discuss some irregular cases that we encountered.

Variations related to programming skills. By a systematic analysis of tables we found that there were not just tables with 99 entries but also tables with 100 entries that were subscripted by product codes. The tables with such an extra entry indicated the use of a defensive style, which is not uncommon in actual Cobol programs. The defense is concerned with code that does not check for the index field to be different from the error code 100. To avoid run-time errors, programmers add an extra entry. In expanding these tables with 100 entries, we would need to preserve the status of an extra entry as well.

Variations related to dead code. In addition to tables that were subscripted by product codes, we were also interested in other tables that involved a field for product codes. We recall our earlier discussion that these tables might require expansion as well. Here is a particularly suspicious example:

```
000431   01   TAB-CONTR.
000432        03   PRODCODE-EL OCCURS 30.
000433             05   PRODCODE              PIC X(02).
000434             05   YEAR-ULTIMO           PIC X(02).
000435             05   CONTR-EL OCCURS 19.
000436                  07  VALDATE-YYDDD     PIC S9(07) COMP-3.
000437                  07  CONTR-PERC        PIC S99V999 COMP-3.
000438                  07  IND-COMP          PIC X(01).
000439             05   IND-EXPIRED           PIC X(01).
000440             05   FILLER                PIC X(05).
```

Notice the PRODCODE field. Also notice that the table has only 30 entries. We examined how this table was used. The table is not indexed by any data item of product-code type of usage, but we found the following loop encoded with GO TO logic.

```
001889   0170.
001890        PERFORM 90-RENT-COMP.
001891        ADD 1 TO SUB-TAB-1.
001892        READ FILE04 INTO PRODCODE-EL (SUB-TAB-1)
001893             AT END COMPUTE CONTR-IN = SUB-TAB-1 - 1
001894                    GO TO 0199.
001895        GO TO 0170.
```

Apparently, this table is filled by retrieving all records from a file called FILE04. There is no evidence that 30 would have been a sufficient number of entries even prior to expansion. During a meeting, a domain expert proposed that this code might be dead. We were also told that the number 30 used to be the original upper limit for product codes when the system was first designed in the 1970s before the upper limit was later extended to 99 product codes. The above table must have been forgotten in this earlier modification effort, which was carried out manually. Since the code happened to be dead already at that time, no program error was ever detected. As a result of the meeting, the customer decided to deal with the eradication of such dead code.

As a side note, such an iterative history of updating is nothing uncommon. For instance, in the view of date expansion: some systems built in 1970 used 1 digit for the year, and were updated in 1980 to deal with 2 digits, were updated to deal with leap years, were updated to deal with the year 2000, were updated to deal with the leap year exception in 2000, were updated to deal with 54 weeks (the year 2000 had 54 weeks), were updated with four digits, are going to be updated in case windowing was used as a Y2K solution, and so on.

Irrelevant tables. There were also tables with OCCURS 99 or 100 in the project that were not expanded because 99 or 100 did not seem to refer to the number of product codes in these cases. This claim was approved by inspections of comments and naming conventions. The following table is an example of a case where table expansion was not needed.

```
00187   01   CI-TABLE.
00188        03   CI-ENTRY OCCURS 99.
00189             05   ENTRY-CI.
00190                  07  CI-REG       PIC 99.
00191                  07  CI-PROV      PIC 99.
00192             05   PART1            PIC X(26).
00193             05   PART2            PIC X(26).
```

Consulting the domain experts, we learned that the name convention CI points to other data than product codes, namely to a certain kind of customer information.

6.7. Specific changes that remained

The transformations of the previous subsections are relatively generic in the sense that they lend themselves to general modification rules. There are some remaining, rather specific changes, which we will explain in an informal manner. The spots were detected using our tools for code exploration, but the changes were performed manually. We stress that even these changes were recorded in patch scripts, which were generated by the `diff` tool, so that we could repeat the changes or adapt the changes. Since we were expecting to make the same changes several times, this is much more cost effective than doing this by hand without recording the changes.

`FILLER` **contraction in group fields.** Picture-string expansion for fields that reside in group fields offers the following complication. If the group field happens to serve as an input/output buffer, or for similar purposes, we might have to preserve the overall length of the group field if possible. That is, we must not exceed the line length of a printer, or a screen. Similarly, we have to avoid data files having to be recreated with a different length. This issue can be addressed by taking advantage of the `FILLER`s that are often part of group fields for buffers and records.[9] In fact, `FILLER`s are used by Cobol programmers both for formatting, and to anticipate data expansion. We measured in some systems that 8% of the physical lines of code consisted of `FILLER`s. Consequently, one can attempt to compensate for an expanded item by reducing the size of a sibling `FILLER`.

The following group field `TABLE-ENTRY` contains a `PRODCODE` field. The group field is 80 characters long, and it used as a line buffer, which is indeed expected to be precisely of this length. The various `FILLER`s are used to optimally format the information to be shown on the screen or to be sent to a line-printer.

```
00225   03   TABLE-ENTRY.
00226        05   FILLER      PIC X(17).
00227        05   PRODCODE    PIC 99.
00228        05   FILLER      PIC X.
00229        05   CODE-NOT    PIC X.
00230        05   MINIMUM     PIC 999V99.
00231        05   FILLER      PIC X(6).
00232        05   COMP-CODE   PIC 99.
00233        05   FILLER      PIC X(46).
```

To retain the length of 80 characters after the expansion of the `PRODCODE` field, we reduce the `FILLER` before `PRODCODE` by one (in fact, any `FILLER` would do):

```
00225   03   TABLE-ENTRY.
00226        05   FILLER     PIC X(16).
00227        05   PRODCODE   PIC 999.
00228        05   FILLER     PIC X.
00229        05   CODE-NOT   PIC X.
00230        05   MINIMUM    PIC 999V99.
00231        05   FILLER     PIC X(6).
00232        05   COM-CODE   PIC 99.
00233        05   FILLER     PIC X(46).
```

[9] Cobol terminology: A `FILLER` declaration takes precisely the form of an ordinary data declaration, except that the keyword `FILLER` is used in the data declaration instead of a data name. This results in dummy (or anonymous) parts of group fields that can only be initialised and not filled with `MOVE` statements.

In this particular example, one could argue that the length of 80 is not exhausted anyway because of the long FILLER at the end of the group field. So even if the length of the group field goes up to 81 after expansion, there will be still 45 unused characters at the tail. In general, such a liberal approach might cause harm, for instance, when such fields are used in subprogram calls, or in file records. The receiving subprogram, e.g., for line-wise printing might insist on 80 character long buffers as for the declaration in the LINKAGE SECTION. Obviously, then a change in the length in one client of the subprogram can have disastrous effects since CALL and LINKAGE SECTION will not fit with each other. Cobol compilers do not even necessarily emit warnings about such a mismatch leading to run-time errors. Hence, in general, we preserved all invariants of the code, including the length of buffers, like the one above.

Updating comments and strings. By simple text search for PRODCODE and other patterns we found that there were also comments and alphanumeric literals that contained PRODCODE. These occurrences often required adaptations. For instance, in some programs, we found DISPLAY paragraphs like the following:

```
00615   0191.
00616        DISPLAY '********************************'.
00617        DISPLAY '** PRODCODE TABLE GREATER THAN 99'.
00618        DISPLAY '********************************'.
00619        CALL 'STOP-4711'.
```

Clearly, the error message is connected to product codes. The message will be outdated after expansion. The new range overflow is at 299 rather than 99. Thus, we manually updated the DISPLAY paragraph:

```
00615   0191.
00616        DISPLAY '*********************************'.
00617        DISPLAY '** PRODCODE TABLE GREATER THAN 299'.
00618        DISPLAY '*********************************'.
00619        CALL 'STOP-4711'.
```

Note that we even added an asterisk "*" in lines 00616 and 00618. We also found affected comments such as the following which prefixed a table declaration indexed by product codes:

```
006000** ELEMENT OCCURS 99 -> PER PRODCODE
```

The comment was converted as follows.

```
006000** ELEMENT OCCURS 299 -> PER PRODCODE
```

One can argue that keeping comments up to date is not worth the effort since they do not affect functional behaviour. There are good reasons why comments should be updated as well. Maintenance programmers do study the comments and if they are confused by inconsistent documentation it will take much more time to do their work properly. The average maintenance programmer is not aware of the entire change history of a system. Hence, he or she cannot easily resolve the inconsistencies when faced with outdated comments. Also when maintenance teams learn that the comments are outdated in some cases, they may assume this to be the case for all comments, including recent comments by their colleagues. This can lead to a considerable loss of productivity.

Strange idioms. The following fragment required a mixture of several modes of expansion. That is, picture-string expansion, literal expansion, and table expansion were relevant at the same time. The fragment hard-codes all possible product codes:

```
00167   01   PRODCODES.
00168        03  FILLER                      PIC X(40)
00169        VALUE '01020304050607080910111213141516171819 20'.
00170        03  FILLER                      PIC X(40)
00171        VALUE '21222324252627282930313233343536373839 40'.
00172        03  FILLER                      PIC X(40)
00173        VALUE '41424344454647484950515253545556575859 60'.
00174        03  FILLER                      PIC X(40)
00175        VALUE '61626364656667686970717273747576777879 80'.
00176        03  FILLER                      PIC X(38)
00177        VALUE '81828384858687888990919293949596979899'.
00178   01   PRODCODE-TABLE  REDEFINES  PRODCODES.
00179        03  PRODCODE  OCCURS 99        PIC X(02).
```

That is, a hard-wired list of all possible product codes is declared using `FILLER`s for blocks of 20 product codes, and the list is redefined to be accessible as a Cobol table with 99 entries. We had to update this declaration to preserve functional behaviour. Eradication of subtoptimal idioms like the one at hand was not an issue in this architectural modification project. So we had to come up with a transformation that upgraded the list of product codes as follows:

```
00167   01   PRODCODES.
00000        03  FILLER   PIC X(48)
00000        VALUE '001002003004005006007008009010011012013014015016'.
00000        03  FILLER   PIC X(48)
00000        VALUE '017018019020021022023024025026027028029030031032'.
00000        03  FILLER   PIC X(48)
00000        VALUE '033034035036037038039040041042043044045046047048'.
00000        03  FILLER   PIC X(48)
00000        VALUE '049050051052053054055056057058059060061062063064'.
00000        03  FILLER   PIC X(48)
00000        VALUE '065066067068069070071072073074075076077078079080'.
00000        03  FILLER   PIC X(48)
00000        VALUE '081082083084085086087088089090091092093094095096'.
00000        03  FILLER   PIC X(48)
00000        VALUE '097098099100101102103104105106107108109110111112'.
00000        03  FILLER   PIC X(48)
00000        VALUE '113114115116117118119120121122123124125126127128'.
00000        03  FILLER   PIC X(48)
00000        VALUE '129130131132133134135136137138139140141142143144'.
00000        03  FILLER   PIC X(48)
00000        VALUE '145146147148149150151152153154155156157158159160'.
00000        03  FILLER   PIC X(48)
00000        VALUE '161162163164165166167168169170171172173174175176'.
00000        03  FILLER   PIC X(48)
00000        VALUE '177178179180181182183184185186187188189190191192'.
00000        03  FILLER   PIC X(48)
00000        VALUE '193194195196197198199200201202203204205206207208'.
00000        03  FILLER   PIC X(48)
00000        VALUE '209210211212213214215216217218219220221222223224'.
00000        03  FILLER   PIC X(48)
00000        VALUE '225226227228229230231232233234235236237238239240'.
00000        03  FILLER   PIC X(48)
00000        VALUE '241242243244245246247248249250251252253254255256'.
00000        03  FILLER   PIC X(48)
```

```
00000         VALUE '257258259260261262263264265266267268269270271272'.
00000         03  FILLER                      PIC X(48)
00000         VALUE '273274275276277278279280281282283284285286287288'.
00000         03  FILLER                      PIC X(33)
00000         VALUE '289290291292293294295296297298299'.
00178    01   PRODCODE-TABLE  REDEFINES  PRODCODES.
00179         03  PRODCODE  OCCURS 299        PIC X(03).
```

Note that the number of product codes covered by each FILLER differs from 20 in the original code to 16 in the adapted code. This deviation is implied by the maximum line length in Cobol. An alternative solution is to use so-called continuation lines, where the alphanumeric literal is then distributed over multiple lines by using the special continuation syntax for Cobol with the dash sign (i.e., "−") in the seventh column.

6.8. Documentation of the changes

Recall that the PRODCODE contract required a detailed documentation of all changes. We addressed this requirement during the design of the solution as follows. We assumed that all changes would be documented by means of source-code annotations including the category of change. One specific way to accomplish this goal was to label each affected line of code by a short conversion code (or marking) using the comment margin starting in column 72 of Cobol files. The annotation starts with a common prefix, namely "PCEXP" (for Product Code EXPansion). Then, one more character is appended to characterise the rule that was applied, and yet another character encoded how the rule was applied. In Fig. 12, we list all codes to be used for annotation. We also illustrate that this discipline of annotation can then be used easily to extract all affected lines in a program. Similarly, we obtained the management summary from Section 5, which lists all changes per category.

Documentation of unchanged patterns. Annotations can also be used to document code fragments that were considered but were not changed, complete with the justification. We do this to alert others to suspicious patterns that are likely to trigger false positives. For instance, when we explored table expansion, we also encountered a table that did not need expansion because it was not at all concerned with product codes (cf. Section 6.6). This sort of non-change can be documented as follows:

```
00187     01   CI-TABLE.
          * NO EXPANSION WAS PERFORMED.                    PCEXPOT
          * THIS TABLE DEALS WITH CUSTOMER INFORMATION.    PCEXPOT
00188           03  CI-ENTRY OCCURS 99.                    PCEXPON
```

The T in PCEXPOT stands for (text) comments. The N in PCEXPON stands for "*negative*" (i.e., no change). The marking of the new comment lines ensures that the added comments are not confused with the primary documentation. These comments help to make explicit that the problem was spotted, not missed, that it once seemed to be a problem, but it is in fact not. In particular when others inspect or test an updated system, this type of documentation turns out to be useful, since it significantly aids in reviewing the completeness of the solution. It is common practice that testers at the customer site will inspect the code, and try to check whether cases were missed. Now they can see when they stumble into suspicious but unchanged parts of the system that we have seen the code, but that no change was necessary. The lines with text comments explain why no change was necessary.

(a)

Rule encodings:

P	Picture-string expansion
O	Table expansion (the O is from OCCURS)
M	Maximum expansion
N	Numeric literal expansion
A	Alphanumeric literal expansion
D	Adaptation of DISPLAY statements
F	FILLER contraction
C	Other adaptations (C stands for *complex*)

Additional remarks:

A	Generic transformation
H	Specific transformation (first by hand, then recorded in a patch script)
T	Comment on transformation
N	Negative: the code was suspicious but no adaptation was needed

(b)

```
> grep PCEXP p7888v15
       05  FILLER              PIC X(16).                  PCEXPFH
       05  PRODCODE            PIC 999.                    PCEXPPA
    03  ELEM01                 OCCURS 299 INDEXED BY IND1. PCEXPOA
       05  PRODCODE            PIC 999.                    PCEXPPA
       05  FK-TK1 OCCURS 300 INDEXED BY IND-TK2  PIC S9(4) PCEXPOA
01  PRODCODE-BOOKING           PIC X(03).                  PCEXPPA
    IF IND1 NOT > 299                                       PCEXPMA
    IF IND1 > 299 GO TO 0191.                               PCEXPMA
     DISPLAY '**********************************'.         PCEXPDH
     DISPLAY '** PRODCODE TABLE GREATER THAN 299'.          PCEXPDH
     DISPLAY '**********************************'.         PCEXPDH
    IF PRODCODE IN WG-MAND-NUM = 000                        PCEXPNA
    IF PRODCODE = '001' OR '002' OR '003' OR '004'          PCEXPAA
...
```

Fig. 12. The modified programs are annotated using PCEXP?? margins. All the possible codes are listed in part (a) of the figure. One can grep for the affected lines as illustrated in part (b) of the figure.

7. Implementation of tools

The described approach to architectural modifications naturally employs tools of the following kind:

- While simple means of code exploration can be based on tools like grep, more advanced tools for program analysis will be ultimately required. These tools involve grammar knowledge to a certain degree, they employ non-trivial algorithms, heuristics, and data structures to compute their results.
- The part of the modification project that allows for generic adaptations can be implemented in terms of automated program transformations. The development of the transformation tools can often be organised as a continuation of earlier efforts on tool support for code exploration and program analysis.
- Manual changes require scripting to record them such that they could be replayed. (In our case, we used ed scripts obtained by the -e option of the Unix tool diff.) Such scripting turns even manual changes into a basic form of automated program transformations.

```
1  if ($current_line =~ /$prodcode_pattern/ ) {
2    extend_the_seed_set($file_name,$line_number,$current_line);
3    if ($declaration) {
4      ++$matrix{column($section)}
5    } elsif ($current_line =~ /(MOVE|SET)/) {
6      ++$matrix{column($1)};
7      propagate($file_name, $line_number, $current_line, $1,
8       "\^\($1|TO\)\$", "", "\^\(ALL|ZERO\(E?\)\(S\?\)|SPACE\(S\?\)\)\$");
9    } elsif ($current_line =~ /(IF)/) {
10     propagate(...);
11   } elsif ($current_line =~ /(READ|WRITE|CALL|...)/) {
12     propagate(...);
13   } else {
14     ++$matrix{column($previous_verb)};
15   }
16 }
17 if ($current_line =~ /(MOVE|SET|IF|READ|WRITE|CALL|...)/) {
18   $previous_verb = $1;
19 }
```

Fig. 13. Excerpt from the `perl` script for code exploration and program analysis.

- While this was not relevant for the PRODCODE project, we have elsewhere encountered the need to develop interactive components that help in walking through affected code locations, approving guesses made by the implemented heuristics, and making selections among possible transformation options.

In principle, tools of the above kind could be developed for use at either the client site, or the service provider's site, or both. We recall that all tools in the PRODCODE project were developed for service purposes. We will now briefly describe the lightweight implementation of the most prominent tools for analysis and transformation that were used in the PRODCODE project. Afterwards, we will discuss general technology issues and options.

7.1. A `perl`-based implementation

In the PRODCODE project, we implemented the required analysis and transformation functionality, as specified in the previous section, in about 1 KLOC of `perl` code. The language `perl` [95,96] is perfectly suited to implement lexical tools for analysis and transformation due to its capabilities for regular expression matching, file handling, and others. We will now discuss noteworthy aspects of the `perl`-based implementation. The purpose of this discussion is not to generally promote the use of `perl` for architectural modifications, but rather to illustrate the characteristics, benefits and limitations of a lightweight approach. Technology options other than `perl` appeared to be more expensive or more risky for use in the PRODCODE project at that time.

Analysis functionality. In Fig. 13, a core fragment of the `perl` script for analysis is shown. The seed set and the list of all affected patterns is produced by this fragment. The script operates on a number of data structures and uses auxiliary functions to deliver the required functionality. It is worth mentioning that this script was initiated during problem analysis, and was later completed and matured to be useful to provide input for the ultimate transformation.

```
1  if (affected($var_at_linenr{$line_number})) {
2    $data_name = $var_at_linenr{$line_number};
3    $picture_position = $vertical_position{$data_name};
4    $old_picture = $vertical_picture{$data_name};
5    $left_part = substr($current_line,0,$picture_position);
6    $right_part = substr($current_line,$picture_position+length($old_picture));
7    $new_picture = map_string($old_picture,2,3);
8    $difference = length($old_picture) - length($new_picture);
9    $right_part = substr($right_part,0,length($right_part) + $difference);
10   $old_line = "$left_margin\*".substr($current_line,1)."$right_margin\n";
11   $new_line = "$left_part$new_picture$right_part";
12   $new_compl_line = "$left_margin$new_line".PCEXPPA."\n";
13 }
```

Fig. 14. Excerpt from the `perl` script for transformation.

We explain the fragment in the figure. In line 1, the `$current_line` of input is checked whether it matches a `$prodcode_pattern`. Recall the conventions `PRODCODE`, `PRDCODE`, or `PC`. If the match succeeds, we `extend_the_seed_set` in line 2. This function records the `$file_name` and the `$line_number` of the `$current_line` where the `$prodcode_pattern` is found. Depending on the location of the matched `$prodcode_pattern`, the counter for the relevant division or section is incremented. If the `$prodcode_pattern` in the `$current_line` was found in a `$declaration` (say, in the `DATA DIVISION`), then it is either in the `LINKAGE` or in the `WORKING STORAGE` `$section`. In line 4 we increment the corresponding counter in `$matrix` by using the type of occurrence as an index for the corresponding `column`. If the `$prodcode_pattern` did not match with a line of `$declaration` code, the match is consequently in the `PROCEDURE DIVISION`. The various statement categories are handled by the `elsif` in lines 5–15. Consider the case where the `$current_line` contains a `$prodcode_pattern` and a `MOVE` or a `SET` statement (lines 5–8). By using the brackets in the pattern `(MOVE | SET)` in line 5, the result of the match, i.e., `MOVE` vs. `SET` is saved in the predefined variable `$1`. In line 6, we count this match by incrementing the `$matrix` according to the verb in the right `column`. Then in line 7, we `propagate` the new variable(s). The function `propagate` extends the set of affected fields. The arguments of `propagate` are listed in lines 7 and 8. These arguments consist of the `$file_name`, the `$line_number`, the `$current_line`, the matched verb (stored in the regexp variable `$1`), plus some line-noise to indicate Cobol syntax that should be skipped in order to arrive at a new Cobol verb.

The cases for `IF` statements and others are handled in a similar way in lines 9–12. The closing `else` in line 13 deals with the case where no verb could be determined in the `$current_line`. This implies that we were in the middle of a multi-line statement started with `$previous_verb` one or more lines ago. In that case, we update the counter using the `$previous_verb` in line 14. Storing the current verb (if any) in `$previous_verb` is accomplished in line 17–19.

Although the `perl` approach is a lexical one, some grammar knowledge is encoded in the implementation. For instance, the list of keywords for recognising the beginning of statements in line 17 in Fig. 13 is directly based on the Cobol grammar.

Transformation functionality. Let us also examine a core fragment of the transformation functionality. The excerpt in Fig. 14 illustrates the line-by-line treatment during

picture-string expansion. The full implementation of the transformation functionality contains a similar part treating table expansion and literal expansion.

The line-wise process of picture-string expansion relies on data structures that we reuse from the analysis functionality. In line 1, we check if an `$affected` variable has been declared in the line at hand. If this is the case, picture-string expansion is attempted. A number of sub-phrases of the corresponding declaration are looked up in lines 2–6, namely the `$data_name` itself, the starting `$picture_position` of the picture mask in the `$current_line`, the `$old_picture` string, and the `$left_part` and `$right_part` before and after the picture string. In line 7, the `$new_picture` string is computed with the function `map_string`. It takes the `$old_picture` string, the old length (which is 2) and the new length (that will be 3) and returns the `$new_picture` string. Picture-string expansion is completed by the construction of the `$new_line` in several steps. In lines 8–9, the `$difference` between the old and new length of the picture mask is determined and the `$right_part` is truncated accordingly. This `$difference` is normally -1, or 0 when a heading `B` (for blank) in a picture string can be used to compensate for the additional digit. In line 10, the `old_line` including the margins is turned into a comment line by concatenating various items, namely the `$left_margin` (columns 1–6), the comment marking "`*`" (column 7), the core of the `$current_line`, the `$right_margin` (columns $>$ 72), and a return (`\n`). In line 11, the `$new_line` of the resulting Cobol program is constructed, which is completed by margins and the annotation `PCEXPPA` in line 12. We recall that the code `PCEXPPA` indicates a picture-string expansion; see Fig. 12.

7.2. Technology issues

We will now discuss technology issues of implementing functionality for source-code analysis and transformation as required in architectural modification projects. Reflecting on such issues is crucial to determine feasibility of projects and to estimate their costs. Here is an overview of some prime issues:

- *Context-free vs. lexical tools*. Grammar-based or context-free technology allows the re-engineer to express analyses and transformations directly in terms of the syntactical patterns of the language at hand. By contrast, the discussed `perl`-based implementation adopts a lexical approach because it basically operates at the level of regular expressions that model sequences of tokens or characters.
- *Comfort of pattern notation*. Users of grammar-based technology can be faced with different notations for encoding syntactical patterns [75]. One option is to directly employ concrete syntactical notation. Another option is to use some term format or tree format or object model. Another dimension of distinction is whether the pattern language comprises all constructs or only a sublanguage (or an abstract syntax), as the result of the normalisation.
- *Support for pretty printing*. Some technologies come readily equipped with a pretty printer for the relevant language; others do not. Not every transformation technology strictly requires a pretty printer to produce readable output. Pretty printing can be pervasive, that is, the result is completely rendered from scratch regardless of the original indentation. Alternatively, pretty printing can be conservative, that is, unchanged parts of the input are not pretty printed [39].

- *Syntax retention*. Transformations should ideally preserve all comments of the input. There are further requirements related to comments. Some modification projects require transforming the comments themselves, or adding comments for documentation. Recall that we encountered these two additional issues in the PRODCODE project. Preservation of comments is just one aspect of syntax retention. We often need to preserve indentation and other layout, too.
- *Domain-specific support*. Pattern matching and building with regard to a language syntax is the simplest kind of domain-specific support for analysis and transformation functionality. Other means of domain-specific support include source-code annotation [48, 77], generic traversal [11,12,58], efficient intermediate representations [13], and suites for disciplined meta-programming.
- *Reusable infrastructure*. The ultimate technology comes readily packaged with reusable infrastructure for parsing, pretty printing, preprocessing, data-flow and control-flow analysis, and others. The development, enhancement, and customisation of such reusable components is an important factor for the eventual provision of a product line for architectural modifications.
- *Industrial strength*. We do not attempt a definition of industrial strength here, but we observe that industrial modification projects on business-critical systems tend to require the following technology attributes:
  - *Scalability and robustness*: The technology must be fit for portfolios in the millions LOC range, and for more complex transformation problems.
  - *Reconfigurability and tolerance*: Adopting the technology to actual parameters like the specific language cocktail at hand must be reasonably simple.

We will now discuss some of the more complex issues in depth. Other issues will be covered later when we consider concrete technology options.

Simplicity of the lexical approach. There are various, widely known tools that allow for the implementation of lexical tools, e.g., `rexx`, `lex`, `perl`, `awk`, and `sed`. The lexical approach is not only conceptually simple, but it is also the difference between failure and success in many cases. There are usually no technical preconditions to embark on the lexical approach. In particular, there is no need for a parser, or a pretty printer. The lexical approach naturally preserves the layout of modified software (cf. syntax retention). Lexical tools can easily operate on source code that involves embedded language constructs or preprocessing constructs. The lexical approach is not just suited for tools for analysis and transformation, but also for special-purpose preprocessors and postprocessors that are needed in many projects. Also, the presentation of newly inserted code (including comments) can be controlled very accurately. Lexical tools are easily deployed at the client site if needed since they usually do not rely on any sophisticated technology.

Limitations of the lexical approach. A problem of lexical tools is the imprecise and unpredictable borderline where these tools stop being effective. Also lexical tools can easily lead to severely damaged converted systems as discussed in [12, Sec. 8.1]. The more complex the code patterns get, the more non-proportional effort has to be spent on recognising and transforming them. For instance, a proper control-flow analysis for Cobol requires the recognition of the nested statement structure, which includes various specific

phrases per statement, and several rules about the meaning of delimiters. This analysis is cumbersome to implement using regular expressions. Another problem is that lexical tools normally tend to reuse grammar knowledge rather poorly, and that they operate at a low level of abstraction. This makes maintenance of these tools more difficult.

Rapid parser development enables context-free tools. Tools that are aware of a language's syntax are, in principle, better suited for the implementation of functionality that involves non-trivial patterns. Unfortunately, such tools are also more expensive to implement. Industrial application of a grammar-based approach requires additional investments like development of parsers for the languages at hand. Even if a parser is available, it is often necessary to tweak it on a per-project basis. Just this tweaking can be more expensive than using a lexical tool and manual work. Language cocktails as they are used in deployed systems, e.g., Cobol + Embedded SQL + CICS transactions + preprocessing, challenge any parsing technology. New technology to rapidly implement parsers has the potential to solve the problem of these expensive up-front investments. We are conducting research in this context [14,46,55,56]. To this end, we combine methods for grammar recovery, grammar deployment, and others. Parser development will also be further simplified by improved parsing technology, e.g., by offering convenient idioms for parser tweaking.

Mix of lexical and context-free tools. Context-free and lexical tools can be used together in a synergistic manner. In such a combination, context-free tools are used for precise analyses as opposed to transformations. This allows us to refrain from investing in pretty printing, layout preservation, and others. Based on the analyses, which include precise data (like exact row and column information), lexical tools are useful to execute the actual modifications. Finally, context-free tools can be used again to approve lexically based modifications, for example, to check that the resulting programs can still be parsed, or that certain postconditions are met. Lexical tools can also be used in other ways to complement context-free tools or to compensate for their weaknesses. Namely, lexical tools can be used to handle preprocessing prior to context-free parsing, and postprocessing to merge the output of a transformation with the original layout, or to resolve layout or documentation problems at a lexical level.

Tolerant parsers. Another combination of the lexical and the context-free approach is to make parsers more tolerant, or less dependent on details of the context-free syntax. This principle underlies fuzzy parsing [47], island grammars [21,66], and tolerant grammars which comprise island and skeleton grammars [46]. These approaches focus on the context-free patterns of interest, while the rest of the input is skipped in a rather lexical style. Tolerant parser development can also be seen as one vital option for rapid parser development, which was identified as the bottleneck of context-free tools above. Tolerant parsers share a problem with lexical tools, i.e., there is a potential of false positives or false negatives. Therefore, we have introduced *skeleton grammars* [46]: these grammars are deduced from a full-blown base-line grammar such that their overall structure and the constructs of interest are inherited.

Reusable infrastructure. Architectural modification projects face various challenges: complex cocktails of languages and platforms, imprecise and incomplete requirements

for changes, fixed pricing, handling of offloading, testing, and others. For such reasons it is important to establish a suitable methodological basis. When one is faced with more complex modifications, the methodological basis needs to be complemented by a technological one. Otherwise the time-line for a precise problem analysis can become unattractive, and the implementation of modification tools involves unaffordable costs and risks. A technological basis comprises a reusable infrastructure for the analysis and transformation of source code; see also [76,82]. Here is a list of components with chances for reuse including elements of methodology in some cases:

- *Preprocessors*. `COPY` books (Cobol) and include files (C/C++) as well as macros need to be expanded, comment columns (Cobol) need to be removed or scaffolded [48,77]. Extra preprocessing can account for a normalisation to eliminate syntactical variation. Preprocessing is challenged by requirements for reversibility (during postprocessing).
- *Postprocessors*. The effects of preprocessing are reversed.
- *Parsers*. The up-front investment is about recovery and development of base-line grammars. These grammars then need to be deployed for the parser technology of choice. We also need processes to rapidly recover or customise grammars and parsers for new dialects and new embedded languages.
- *Pretty printers*. Given a grammar of a language, an initial pretty printer can be generated. A proper investment is about the development of a customised pretty printer. The pretty print rules should also be reusable for conservative pretty printing of just those patterns that are affected by problem-specific transformations.
- *Name resolvers*. All use sites of identifiers are readily connected to declaration sites. Additional properties can be provided such as the number of use sites, or the storage space for any field including group fields in Cobol.
- *Data-flow analysers*. These components serve standard inquiries such as the uses that are reachable from a definition. Other analysers deal with type-of-usage analysis as in the PRODCODE project. Further analysers compute data for the implementation of problem-specific refactoring and slicing tools.
- *Control-flow analysers*. These components help in performing general-purpose and problem-specific control-flow restructuring as well as dead-code detection. These analysers can also be used in projects for wrapping legacy components, for extracting business rules, and for replacing transaction management and others.
- *Low-level tooling*. This concerns plug-in technology to extend editors by facilities for structured editing and interactive transformations. It also concerns infrastructure for building and testing modification tools. Furthermore, it concerns technology for component-based modification tools.

There is a tension between implementing ad hoc solutions and investing into reusable components. Consider, for example, name resolution. To develop a reusable component for full name resolution just for the PRODCODE project is economically not justified. This is based on the size of the project and the relatively small budget for the modification project. On the other hand, for projects in the multi-million LOC range, investment into building a reusable component for name resolution is likely to pay off. There is also a tension between precise components vs. leaving room for heuristics. For instance, type-of-usage analysis amounts to a precise algorithm for some part, but the attack of pollution problems

naturally involves heuristics. For instance, in the PRODCODE project, we had to uncover buffer-like fields on the basis of fuzzy criteria for extra-large types. This implies that reusable components should be sufficiently parameterised, and that the accommodation of heuristics should be convenient.

7.3. Technology options

There are various technologies of potential use for architectural modifications: parsing generators, transformation environments, meta-programming frameworks, program slicers, and all kinds of generic language technology [10]. We can also categorise in terms of industrial strength versus home grown systems, or in terms of lightweight or sophisticated technology, and combinations of all these. In addition to reusing existing technology, one often ends up implementing ad hoc solutions, at least for parts of the problem. Listing, categorising, and assessing all technology options and specific technologies is beyond the scope of this paper, but we discuss a few illustrative experiences with some technology options.

The PRODCODE benchmark. We will now discuss a few technology options while using the PRODCODE problem as a basis for assessment. In fact, the PRODCODE problem has actually served as a kind of internal benchmark not just for us, but also for close colleagues. The PRODCODE project is suitable for such a comparative assessment:

- it is a real-world case that is of reasonable size,
- the specification of the entire solution is well documented,
- the original system and the converted one were available as a reference.

The three options that follow take a context-free approach to solving the problem, as opposed to our lexical implementation. The first option is the ASF + SDF Meta-Environment. The second option is Haskell-based transformation technology. The third option is strategic term rewriting with Stratego.

ASF + SDF Meta-Environment. The ASF + SDF Meta-Environment [15,44] bundles generic language technology. Most notably, it provides a couple of formalisms for executable specifications: SDF (*Syntax Definition Formalism*) and ASF (*Algebraic Specification Formalism*). SDF is a rich syntax formalism which is supported by a Generalised LR parser generator. ASF can be viewed as a typed programming language for conditional term rewriting using concrete syntax in the rules. Recall that we used ASF + SDF rewrite rules for the development of the specification in Section 6. In the actual project, we developed this specification back to back with the lexical tools for code exploration, analysis, and modification. The specification only served for a concise presentation of the solution, and it was not meant as an actual implementation. This status was implied by the following technology issues. Firstly, at that time we did not have a Cobol front-end that was immediately fit for parsing the PRODCODE code. Because of the size and the time-line of the PRODCODE project, we had only a few days to complete the problem analysis. This issue was sufficient to trigger the deployment of lexical tools. Also, some side conditions would have been more complicated with the version of the ASF+SDF Meta-Environment that was available at that time. This concerns layout preservation and

```
pic99To999 :: Names -> CblPrg -> CblPrg
pic99To999 affected = fold alg1
  where
    alg1 = idmap {
                   f_Data_desc = \x (_,n,_) ->
                    if n `elementOf` affected then fold alg2 x
                                              else x }
    alg2 = idmap {
                   f_Pic_clause = \x mask ->
                    case mask of
                     "99" -> Pic_clause "999"
                     _    -> x }
```

Fig. 15. Picture-string expansion in Haskell (simplified). The top-level function `pic99To999` converts a given Cobol program while being parameterised in the names of affected fields. The picture-string expansion is encoded as nested `fold` (say, traversal) over the program. The first traversal (see `alg1`) stops at the level of data-field declarations. The second traversal (see `alg2`) stops at the `PIC` clause, and replaces the picture string. The traversals are obtained by refining the identity map (cf. `idmap`) to handle specific patterns `f_Data_desc` and `f_Pic_clause`.

documentation of changes. The present status of the ASF + SDF Meta-Environment and our infrastructure has improved considerably. Firstly, we have recovered several grammars, e.g., the IBM VS Cobol II grammar [54], and we improved the process to deploy parsers [45,49]. Secondly, a simple form of layout preservation is transparently supported by the ASF + SDF Meta-Environment [9]. There is also work underway to improve the reconfigurability of pretty printing. Thirdly, the ASF + SDF Meta-Environment provides meanwhile very convenient support for generic traversal, which makes it easy to rewrite complex parse trees [11,12]. For a newer case study, which demonstrates the current infrastructure, we refer to [91]. In that paper, the ASF + SDF Meta-Environment is employed in a complicated Cobol case study in which `GO TO` statements were removed, and the control-flow was improved in several millions LOC Cobol of different dialects.

Haskell-based language processing. Functional programming languages like SML or Haskell are quite suitable for implementing functionality for analysis and transformation. Firstly, the concept of algebraic data types together with term matching and building is a good fit for the typed manipulation of program representations. Secondly, the available abstraction mechanisms facilitate concise and reusable implementations of complex problems. This is illustrated with generic refactorings in [53]. The PRODCODE project served as a running example for developing an architecture for Haskell-based transformation systems in [50]. The paper clarifies that by integrating external components such as industrial-strength parsers and adding support for generic traversal, functional programming becomes a viable platform for re-engineering; see also [57]. Further experiences with using functional programming for processing deployed systems are reported by others in [24]—that time being based on SML and being concerned with the Y2K problem. In Fig. 15, we show the Haskell implementation of the picture-string expansion. This encoding relies on the traversal generator Tabaluga [50]. One might argue that pattern matching and construction is not very readable. This is implied by the use of prefix terms as opposed to concrete syntactical notation. The requirement for layout preservation was met by operating on an algebraic data type for parse trees with extra

positions for layout information. These positions are elided in Fig. 15. As in the case with the ASF + SDF Meta-Environment, documentation of changes had to be left to a separate postprocessor because the line-wise and column-sensitive style of documentation is not easily accomplished at the level of parse trees.

Stratego. Stratego [94] is a language supporting strategic rewriting where one can define concrete rewrite rules, normalisation strategies (such as innermost), and traversal strategies (such as top-down and bottom-up traversals). Stratego is integrated with the powerful parsing technology of the ASF + SDF Meta-Environment. In [97], analysis and transformation functionality for the PRODCODE project was reconstructed with Stratego using our specification (as sketched in Section 6) as input. The problem of layout preservation was attacked in Stratego so-called overlays [93]. This feature supports term access using alternative signatures; in this case, the alternative signature hides positions for layout. The original contribution of Stratego is the liberty to easily compose different traversal schemes. This liberty is not immediately needed for straightforward modifications, where a small number of fixed traversal strategies appear to be sufficient. Free-wheeling traversal schemes are eventually of use when implementing advanced and parametric functionality for analysis and transformation; see [11,53,58] for a discussion.

An open-ended list of options. We adopt the following criteria. Firstly, we only consider systems that are not tied to a specific source language. Language-specific systems, e.g., transformation frameworks for fixed object languages are nevertheless useful for specific projects, e.g., Siber Systems' CobolTransformer [80]. Furthermore, we only consider systems that cover both parsing and transformation as opposed to merely parser generators, or attribute-grammar systems, which are less convenient for transformation problems. Finally, we focus on industrial-strength systems using a pragmatic definition: the system must have been used in several industrial modification projects with a range of different parameters carried out by the system developers, or preferably also by a licensee of the system. The following systems meet all these criteria: DMS [79], RainCode [71], Refine [51], TXL [20]. Front-ends for a number of typical languages for business-critical software are available for these systems. All the systems at least support manipulation of source-code representations—normally a rule-based language akin to ASF + SDF.

This ends our discussion of the PRODCODE project. This discussion comprised the analysis of the problem, the project economics, the design of the solution, and the implementation of tools.

8. Concluding remarks

In this paper, we have elaborately discussed the issue of deployed systems hitting their architectural borders. The malleability of such systems then needs to be revitalised by architectural modification projects.

We have clarified that architectural modification projects are important: deployed systems are normally business critical; enabling their change is vital to preserve the associated assets. We have argued that the need for architectural modifications is a reality of software evolution. This is because there are always new forms of software asbestos that unintentionally invade software. We have discussed the project drivers of such modification

efforts, which need to be understood to successfully complete modification projects. To this end, we have elaborately discussed one specific project in full detail. We have indicated that this type of project, when done manually, is a high-risk and high-cost effort. We have shown how the development of a precise problem specification helps in the course of software modification projects. To this end, source-code exploration is a vital method. We have discussed cost estimation and contracting. We have illustrated that the technical solution of a modification project is typically designed by means of different kinds of rules and code samples. We have discussed tool implementation and service delivery.

To summarise, we have provided detailed insight in the nuts and bolts of architectural modification efforts, and we have delivered a road-map for computer-aided life-cycle enabling for software. Others can use our work or a variant thereof to conduct architectural modification efforts for their own deployed software systems, when malleability of these systems is not in alignment with business needs. The success of a large-scale modification effort is dependent on a systematic problem analysis, on a high degree of automation, and on the choice of the appropriate technology, but also on getting the system owner involved in the right way such that managed and automated modification can replace hand-crafted maintenance.

Acknowledgements

This research has been partially sponsored by the Dutch Ministry of Economic Affairs via contract SENTER-TSIT3018 *CALCE: Computer-Aided Life Cycle Enabling of Software Assets*. We are grateful for the very detailed and constructive comments that we received from the three anonymous referees of the "Science of Computer Programming" journal. We are also grateful for the opportunity to discuss a real-world project that we carried out for an anonymous customer.

References

[1] E. Arranga, I. Archbell, J. Bradley, P. Coker, R. Langer, C. Townsend, M. Weathley, In Cobol's defense, IEEE Software 17 (2) (2000) 70–72, 75.
[2] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, 2nd edition, Addison-Wesley, 2003 (1st edition appeared in 1998).
[3] P.G. Bassett, Framing Software Reuse, Yourdon Press, Prentice-Hall, 1996.
[4] I. Baxter, M. Mehlich, Preprocessor conditional removal by simple partial evaluation, in: P. Aiken, E. Burd, R. Koschke (Eds.), Proceedings; Working Conference on Reverse Engineering, WCRE, IEEE Computer Society, 2001, pp. 291–300.
[5] L.A. Belady, M.M. Lehman, A model of large program development, IBM Systems Journal 15 (3) (1976) 225–252.
[6] B. Boehm, Software Engineering Economics, Prentice-Hall, 1981.
[7] B. Boehm, B. Clark, E. Horowitz, C. Westland, R. Madachy, R. Selby, Cost models for future life cycle processes: COCOMO 2.0, Annals of Software Engineering 1 (1995) 57–94.
[8] J. Bosch, J. van Gurp, Explicit modelling of architecture design decisions, in: Slides Dagstuhl seminar 03061 Software Architecture: Recovery and Modelling, February, 2003.
[9] M.G.J. van den Brand, J.J. Vinju, Rewriting with layout, in: N. Derschowitz, C. Kirchner (Eds.), Proceedings of the First International Workshop on Rule-Based Programming, September, 2000.
[10] M.G.J. van den Brand, P. Klint, C. Verhoef, Re-engineering needs generic programming language technology, ACM SIGPLAN Notices 32 (2) (1997) 54–61.

[11] M.G.J. van den Brand, P. Klint, J.J. Vinju, Term rewriting with traversal functions, ACM Transactions on Software Engineering and Methodology 12 (2) (2003).
[12] M.G.J. van den Brand, M.P.A. Sellink, C. Verhoef, Generation of components for software renovation factories from context-free grammars, Science of Computer Programming 36 (2–3) (2000) 209–266.
[13] M.G.J. van den Brand, H.A. de Jong, P. Klint, P.A. Olivier, Efficient annotated terms, Software—Practice and Experience 30 (2000) 259–291.
[14] M.G.J. van den Brand, A.S. Klusener, L. Moonen, J.J. Vinju, Generalized parsing and term rewriting: Semantics driven disambiguation, in: B.R. Bryant, J. Saraiva (Eds.), Proceedings of the Third Workshop on Language Descriptions, Tools and Applications, LDTA'2003, Electronic Notes in Theoretical Computer Science, vol. 82, 2003.
[15] M.G.J. van den Brand, J. Heering, H. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J.J. Vinju, E. Visser, J. Visser, The ASF + SDF meta-environment: a component-based language development environment, in: Compiler Construction 2001 (CC 2001), LNCS, Springer, 2001.
[16] P.J. Brown, Macroprocessors and Techniques for Portable Software, John Wiley and Sons, 1974.
[17] G.D. Brown, Cool compiler directing statements in the new standard. CobolReport.com, June, 2000, http://www.cobolreport.com/columnists/gary/06122000.htm.
[18] G.D. Brown, Typing data in the new COBOL standard. CobolReport.com, February, 2001, http://cobolreport.com/columnists/gary/02262001.htm.
[19] Y. Chae, S. Rogers, Successful COBOL Upgrades: Highlights and Programming Techniques, John Wiley and Sons, 1999.
[20] J.R. Cordy, C.D. Halpern-Hamu, E. Promislow, TXL: A rapid prototyping system for programming language dialects, Computer Languages 16 (1) (1991) 97–107.
[21] A. van Deursen, T. Kuipers, Building documentation generators, in: H. Yang, L. White (Eds.), Proceedings; IEEE International Conference on Software Maintenance, ICSM, IEEE Computer Society Press, 1999, pp. 40–49.
[22] A. van Deursen, L. Moonen, Type inference for COBOL systems, in: I. Baxter, A. Quilici, C. Verhoef (Eds.), Proceedings; Working Conference on Reverse Engineering, WCRE, IEEE Computer Society Press, 1998, pp. 220–230.
[23] J. Dueñas, W. Lopes de Oliveira, J. de la Puente, Architecture recovery for software evolution, in: P. Nesi, F. Lehner (Eds.), Proceedings: 2nd Euromicro Conference on Software Maintenance and Reengineering, IEEE Computer Society Press, 1998, pp. 113–120.
[24] P.H. Eidorff, F. Henglein, C. Mossin, H. Niss, M.H. Sørensen, M. Tofte, AnnoDomini: from type theory to Year 2000 conversion tool, in: ACM99 [70], pp. 1–14.
[25] D. Faust, C. Verhoef, Software product line migration and deployment, Software: Practice & Experience 33 (2003) 933–955.
[26] J.M. Favre, The CPP paradox, in: Proceedings of the 9th European Workshop on Software Maintenance, DURHAM'95, 1995.
[27] J.M. Favre, Preprocessors from an abstract point of view, in: S.A. Bohner, A. Cimitile (Eds.), Proceedings; IEEE International Conference on Software Maintenance, ICSM, IEEE Computer Society Press, Washington, 1996, pp. 329–339.
[28] G. Florijn, C. Baars, Experiences with architecture (in Dutch: Ervaringen met architectuur), May, 2003, CIBI–CERC seminar, Utrecht, The Netherlands, http://www.cibit.nl/site.nsf/p/Nieuws-Seminars-Architectuur-13_mei_-_Ev%ent_'Ervaringen_met_Architectuur_
[29] M. Fowler, When to make a type, IEEE Software (2003) 12–13.
[30] M. Fowler, Who needs an architect? IEEE Software (2003) 11–13.
[31] M. Fowler, Platform independent malapropism, September, 2003, M. Fowler's Bliki, http://www.martinfowler.com/bliki/PlatformIndependentMalapropism.html.
[32] B. Hall, Year 2000 tools and services, in: Symposium/ITxpo 96, The IT Revolution Continues: Managing Diversity in the 21st Century, GartnerGroup, 1996.
[33] D.R. Harris, H.B. Reubenstein, A.S. Yeh, Reverse engineering to the architectural level, in: D. Perry (Ed.), Proceedings of the 17th Proc. International Conference on Software Engineering, IEEE Computer Society Press, 1995.
[34] Information technology—Programming languages, their environments and system interfaces—Programming language COBOL, Standard, ISO/IEC FCD 1989:2002.

[35] C. Jones, Applied Software Measurement: Assuring Productivity and Quality, 2nd edition, McGraw-Hill, 1996.
[36] C. Jones, The Year 2000 Software Problem—Quantifying the Costs and Assessing the Consequences, Addison-Wesley, 1998.
[37] N. Jones, Year 2000 market overview, Technical Report, GartnerGroup, Stamford, CT, USA, 1998.
[38] C. Jones, Software Assessments, Benchmarks, and Best Practices, Addison-Wesley, 2000.
[39] M. de Jonge, Pretty-printing for software reengineering, in: G. Antoniol, I. Baxter (Eds.), Proceedings; IEEE International Conference on Software Maintenance, ICSM, IEEE Computer Society Press, 2002, pp. 550–559.
[40] T.W. Keller, Change costing in a maintenance environment, in: S.A. Bohner, A. Cimitile (Eds.), Proceedings; IEEE International Conference on Software Maintenance, ICSM, IEEE Computer Society, 1996 (In lecture notes accompanying the keynote address).
[41] B.W. Kernighan, D.M. Ritchie, The C Programming Language, Prentice-Hall, 1978.
[42] W.M. Klein, OldBOL to NewBOL: A COBOL Migration Tutorial for IBM, Merant Publishing, 1998.
[43] A. Kleppe, J. Warmer, W. Bast, MDA Explained: The Model Driven Architecture: Practice and Promise, Addison Wesley, 2003, p. 192.
[44] P. Klint, A meta-environment for generating programming environments, ACM Transactions on Software Engineering and Methodology 2 (2) (1993) 176–201.
[45] P. Klint, R. Lämmel, C. Verhoef, 2003. Towards an engineering discipline for grammarware, http://www.cs.vu.nl/grammarware/ (submitted for publication).
[46] A.S. Klusener, R. Lämmel, Deriving tolerant grammars from a base-line grammar, in: S. Lawrence Pfleeger, C. Verhoef (Eds.), Proceedings; IEEE International Conference on Software Maintenance, ICSM, IEEE Computer Society, 2003, pp. 179–188.
[47] R. Koppler, A systematic approach to fuzzy parsing, Software Practice and Experience 27 (6) (1997) 637–649.
[48] J. Kort, R. Lämmel, Parse-tree annotations meet re-engineering concerns, in: Proc. Source Code Analysis and Manipulation, SCAM'03, IEEE Computer Society Press, 2003, pp. 161–172.
[49] J. Kort, R. Lämmel, C. Verhoef, The grammar deployment kit, in: M.G.J. van den Brand, R. Lämmel (Eds.), Proc. Language Descriptions, Tools, and Applications, LDTA'02, ENTCS, vol. 65, Elsevier Science, 2002, p. 7.
[50] J. Kort, R. Lämmel, J. Visser, Functional transformation systems, in: 9th International Workshop on Functional and Logic Programming, Benicassim, Spain, July 2000, Technical University of Valencia, Publication 2000/2039, Valencia, UPV University Press, 2000 (September).
[51] G. Kotik, L. Markosian, Application of REFINE language tools to software quality assurance, in: Proc. of the 9th Knowledge-Based Software Engineering Conference, KBSE'94, Monterey, CA, 1994, p. 4.
[52] R. Lämmel, Object-oriented COBOL: concepts & implementation, in: Jon Wessler (Ed.), COBOL Unleashed, Macmillan Computer Publishing, 1998, p. 44.
[53] R. Lämmel, Towards generic refactoring, in: Proc. ACM SIGPLAN Workshop on Rule-based Programming, ACM Press, 2002, pp. 15–28.
[54] R. Lämmel, C. Verhoef, VS COBOL II grammar Version 1.0.3, 1999, http://www.cs.vu.nl/grammars/browsable/vs-cobol-ii/.
[55] R. Lämmel, C. Verhoef, Cracking the 500-language problem, IEEE Software (2001) 78–88.
[56] R. Lämmel, C. Verhoef, Semi-automatic grammar recovery, Software—Practice & Experience 31 (15) (2001) 1395–1438.
[57] R. Lämmel, J. Visser, A Strafunski application letter, in: V. Dahl, P. Wadler (Eds.), Proc. of Practical Aspects of Declarative Programming, PADL'03, LNCS, vol. 2562, Springer-Verlag, 2003, pp. 357–375.
[58] R. Lämmel, E. Visser, J. Visser, 2004, The essence of strategic programming, Draft, http://www.cwi.nl/~ralf/.
[59] M.M. Lehman, Laws of program evolution—rules and tools for programming management, in: Proceedings of the Infotech State of the Art Conference, Why Software Projects Fail, Pergammon Press, 1978, pp. 11/1–11/25.
[60] M.M. Lehman, Laws of software evolution revisited, in: C. Montangero (Ed.), Software Process Technology, EWSPT 96, LMCS, vol. 1149, Springer-Verlag, Nancy, France, 1996, pp. 108–124.

[61] P. Livadas, D. Small, Understanding code containing preprocessor constructs, in: Third International Workshop on Program Comprehension, IEEE Computer Society Press, 1994, pp. 89–97.
[62] M.W. Maier, E. Rechtin, The Art of Systems Architecting, 2nd edition, CRC Press, 2000.
[63] B. Manachem, Software quality, producing practical and consistent software, Technical Report, GartnerGroup, Stamford, CT, USA, 1997.
[64] S. McConnell, Code Complete, Microsoft Press, 1993.
[65] S. McConnell, Rapid Development, Microsoft Press, 1996.
[66] L. Moonen, Generating robust parsers using island grammars, in: P. Aiken, E. Burd, R. Koschke (Eds.), Proceedings; Working Conference on Reverse Engineering, WCRE, IEEE Computer Society Press, 2001, pp. 13–22.
[67] OMG Model Driven Architecture, 2003. http://www.omg.org/mda/.
[68] R. van Ommering, Principle software architect, Philips Research Eindhoven, the Netherlands, May, 2001 (personal communication).
[69] Ovum Ltd, Report on the Status of Programming Languages in Europe, Ovum Report, London, 1997.
[70] Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of programming languages, POPL'99, January 20–22, 1999, San Antonio, TX, ACM SIGPLAN Notices, ACM Press, New York, USA, 1999.
[71] Rain Code Company, RainCode Engine, 2004. http://www.raincode.com/engine.html.
[72] G. Ramalingam, J. Field, F. Tip, Aggregate structure identification and its application to program analysis, in: ACM99 [70], pp. 119–132.
[73] A. Robbins, UNIX in a Nutshell: System V Edition, 3rd edition, O'Reilly & Associates Inc., 1999.
[74] D. Schricker, Data pointers, CobolReport.com, 2000, http://cobolreport.com/columnists/don/08142000.htm.
[75] M.P.A. Sellink, C. Verhoef, Native patterns, in: M. Blaha, A. Quilici, C. Verhoef (Eds.), Proceedings; Working Conference on Reverse Engineering, WCRE, IEEE Computer Society Press, 1998, pp. 89–103.
[76] M.P.A. Sellink, C. Verhoef, An architecture for automated software maintenance, in: D. Smith, S.G. Woods (Eds.), Proceedings of the Seventh International Workshop on Program Comprehension, IEEE Computer Society Press, 1999, pp. 38–48.
[77] M.P.A. Sellink, C. Verhoef, Scaffolding for software renovation, in: J. Ebert, C. Verhoef (Eds.), Proc. Conference on Software Maintenance and Reengineering, CSMR'00, IEEE Computer Society Press, 2000, pp. 161–172.
[78] M.P.A. Sellink, H.M. Sneed, C. Verhoef, Restructuring of COBOL/CICS legacy systems, in: P. Nesi, C. Verhoef (Eds.), Proceedings of the Third European Conference on Maintenance and Reengineering, IEEE Computer Society Press, 1999, pp. 72–82.
[79] Semantic Designs Incorporated, The DMS Software Reengineering Toolkit, 2004, http://www.semdesigns.com/Products/DMS/DMSToolkit.html.
[80] Siber Systems Inc., CobolTransformer—Peek Under the Hood: Technical White Paper, 1997, http://www.siber.com/sct/tech-paper.html.
[81] Siber Systems Inc., MF, IBM, I-Cobol To Fujitsu Cobol Converters, 2004, http://www.siber.com/sct/tools/2fsc.html.
[82] H.M. Sneed, Architecture and functions of a commercial software reengineering workbench, in: P. Nesi, F. Lehner (Eds.), Proc. 2nd Euromicro Conference on Software Maintenance and Reengineering, IEEE Computer Society Press, 1998, pp. 2–10.
[83] H.M. Sneed, Objektorientierte Softwaremigration, Addison-Wesley, 1998 (in German).
[84] H.M. Sneed, Risks involved in reengineering projects, in: F. Balmas, M. Blaha, S. Rugaber (Eds.), Proceedings; Working Conference on Reverse Engineering, WCRE, IEEE Computer Society Press, 1999, pp. 204–211.
[85] Software Engineering Institute, Carnegie Mellon University, How Do You Define Software Architecture? 2004, http://www.sei.cmu.edu/architecture/definitions.html.
[86] S. Some, T. Lethbridge, Parsing minimization when extracting information from code in the presence of conditional compilation, in: Sixth International Workshop on Program Comprehension, IEEE Computer Society Press, 1998, pp. 118–125.
[87] H. Spencer, G. Collyer, #ifdef considered harmful, or portability experience with C news, in: USENIX Conference, 1992.
[88] T. Spitta, F. Werner, Die Wiederverwendung von Daten in SAP R/3, Information Management & Consulting (IM) 15 (2000) 51–56 (in German).

[89] A.A. Terekhov, Recovery and improvement of a software architecture: a case study, February, 2003, Slides Dagstuhl seminar 03061 Software Architecture: Recovery and Modelling.
[90] A.A. Terekhov, C. Verhoef, The realities of language conversions, IEEE Software 17 (6) (2000) 111–124.
[91] N.P. Veerman, Revitalizing modifiability of legacy assets, Journal of Software Maintenance and Evolution (Special Issue on CSMR 2003) 16 (4–5) (2004) 219–254.
[92] C. Verhoef, Quantitative IT portfolio management, Science of Computer Programming 45 (1) (2002) 1–96.
[93] E. Visser, Strategic pattern matching, in: P. Narendran, M. Rusinowitch (Eds.), Rewriting Techniques and Applications, RTA'99, Trento, Italy, July, 1999, LNCS, vol. 1631, Springer-Verlag, 1999, pp. 30–44.
[94] E. Visser, Z.-A. Benaissa, A. Tolmach, Building program optimizers with rewriting strategies, in: International Conference on Functional Programming, ICFP'98, Baltimore, Maryland, September 1998, ACM SIGPLAN, 1998, pp. 13–26.
[95] L. Wall, R.L. Schwartz, Programming Perl, O'Reilly & Associates Inc., 1991.
[96] L. Wall, T. Christiansen, R.L. Schwartz, Programming Perl, 2nd edition, O'Reilly & Associates Inc., 1996.
[97] H. Westra, CobolX: transformations for improving COBOL programs, in: Proc. Second Stratego User's Day, 2001, Technical Report, Utrecht University.
[98] R. Widmer, COBOL Migration Planning, Edge Information Group, 1998.