Understanding Legacy Architectures

Arie van Deursen
CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

November 15, 2001

Abstract

This paper looks at architecture from the perspective of legacy systems. Legacy systems generally do not have a documented architecture, which complicates dealing with many of the problems pertaining to legacy systems. Extracting architectural structures from a legacy system and presenting them in a coherent hypertext framework can provide the architectural understanding needed to deal with these problems. A starting point is to compare the development view present in naming conventions with the logical view that can be extracted from the source. Component interfaces can be described using types extracted from the source; Novel componentizations can be obtained using grouping techniques such as cluster and concept analysis. The various architectural structures studied are illustrated using several example mainframe-based legacy systems.

Note Work sponsored by the *Telematica Instituut*, Enschede, The Netherlands; Project *Domain-Specific Languages*.

1 Introduction

In this paper, we will be concerned with software architecture for legacy systems. Following [3], a legacy information system is any information system that significantly resists modification and evaluation to meet new and constantly changing business requirements. Legacy information systems share many negative characteristics. The most typical ones include [3]:

- Legacy systems consist of millions of lines of code.
- They are geriatric, often more than 10 years old.
- They are written in languages like Cobol.
- They are built around a legacy database service, such as IMS, or sometimes do not even use use a database management system at all.
- Legacy systems are autonomous. They operate independently, with little or no interface with other applications.

In short, legacy systems are inflexible, as well as *brittle*: they are easily broken when changed. Problems arising from legacy systems include failures, maintenance, inappropriate functionality, lack of documentation, and poor performance. The costs involved for a single legacy system often exceed hundreds of millions of dollars per year, and these costs must be made, as legacy systems typically are *mission critical*: they must be operational at all times [3].

Many of the problems of these legacy systems are due to the fact that the system's architectures are virtually undocumented. Moreover, these architectures have degraded over time, due to modifications or extensions that violate the initial architectural principles, resulting in overall chaos.

In this paper, we will explore what we can do to alleviate these problems. We will investigate what software architecture can mean for legacy systems (Section 2), which aspects of architecture can be recovered (Section 3), and how the resulting architectural structures can be presented in a concise, comprehensible and coherent manner (Section 5). Moreover, we will look at various architectural structures in more detail (Section 6). In particular, we will look at the use of *types* to document legacy interfaces (Section 7), and the use of *concept analysis* to reorganize existing systems in an object-oriented manner (Section 8). The concepts, tools and techniques covered are illustrated using examples taken from our experience in analyzing the architecture of a range of real life Cobol legacy systems.

2 Motivation

Having an explicit software architecture available is as valuable for legacy systems as it is for systems whose construction has just been initiated ("architecture-based software development"). Situations where architectures for legacy systems are required include:

- The functionality of a legacy system should be made available via different channels: the typical example is web-enabling the back-office systems of banks or insurance companies;
- Company *A* and *B* consider a merger (or acquisition). Should the software systems of *A* or *B* be used in the new company? What is their quality? Are the system architectures sufficiently flexible to leverage the new business opportunities created by the merger?
- A software house specialized in outsourcing offers fixed-price maintenance services for legacy systems. When bidding for a new contract, it needs to determines a price, based on an architectural review and an assessment of the capabilities of the legacy system to deal with the planned future changes.
- New people (aptly called "software immigrants" by [26]) are brought into the team responsible for maintaining a legacy system. Before they can start being productive, they need to acquire an understanding of the legacy system's software architecture.
- Software maintenance of a particular legacy system is experienced as too difficult and error-prone by both management and the software maintenance team. To remedy this, a renovation of the legacy system is planned. A key element of the renovation is an architectural review, including an assessment of the architecture's capabilities to deal with the anticipated future modification requests.

In one of our collaborations, a customer used the metaphor of renovating a city to illustrate the importance of architecture. When bidding for a renovation contract, we need to know whether we are planning to rebuild a city like Amsterdam (750,000 inhabitants, historic center) or a village such as Stovepipe Wells (2 inhabitants, extreme weather conditions of Death Valley). Moreover, we need to know where to put up fences so that we can safely reconstruct the houses behind them. We may have to redirect traffic in order to do this, making sure that we don't introduce unresolvable congestion, and reorganize infrastructure such as electricity, water, sewer, and cable television. We won't be able to change much to the city's layout (architecture), but we have to be fully aware of it when we plan any city modification.

The same holds for legacy architectures: it is unlikely that we can drastically change a system's architecture, but before conducting any change we have to fully understand what we can do to isolate system parts so that we can safely change them, and what the anticipated effects of our changes are.

3 Software Architecture

In this paper, we adopt the definition of software architecture as proposed by Bass *et al* [1, pp.23–24]:

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

In other words, we distinguish system components as well as their interfaces (the externally visible properties), and the ways in which components depend on each other via these interfaces. Hiding details behind interfaces yields an *abstraction* of the underlying system. The level of abstraction chosen, and the granularity of the components, depends on the purpose of the architecture: assessing the impact of a particular use case to be implemented, for example, will require a different view than analyzing whether the given subdivision into components results in optimal work assignments for programming teams. This same freedom in choosing the level of abstraction is present when searching for legacy architectures: the anticipated use of an extracted architecture affects the way in which components and their interfaces are defined.

The given definition implies that every system has an architecture [1]. There may be very few components, or the given subdivision into components may be far from optimal, but some architecture must exist. The existing architecture, however, may be *unknown* to the development team, i.e. not documented in any way. The original designers may be gone, or the initial architecture may have been degraded beyond recognition, and the effect is that no one really understands the system's architecture. Architecture extraction helps to make forgotten architectural structures explicit.

This definition also emphasizes that one system can comprise more than one structure. No one structure holds the irrefutable clam to being *the* architecture. Example *architectural structures* listed by [1] include the module structure (for work assignments), the logical structure (how functions share data), the process structure (how programs run concurrently), and the call structure (how procedures invoke each other with parameters). In our search for architectures for legacy systems, we will encounter these and several other architectural structures.

The observation that a software architecture is a mix of structures and views, used for various purposes, is made explicit by the "4+1 View Model" of architecture proposed by Kruchten [16]. This model describes software architecture using five concurrent views:

- The *logical* view describes the design's object model, or entity relationship diagram;
- The *process* view deals with the design's concurrency and synchronization aspects;
- The *physical* view covers the mapping of the software onto hardware;
- The *development* view describes the software's static organization in its development environment.

The 5th element of this model consists of a selected set of use cases (scenarios) to illustrate the other views.

For the purpose of legacy system architecture extraction, we will mainly look at the logical and development view, and to some extent at the process view. In fact, the development and logical view are very close. Kruchten observes, however, that the larger the project, the greater the distance between these views. Development structuring is constrained by team organization, expected magnitude of code, degree of expected reuse and commonality, layering policies, release policy, and configuration management [16]. Thus, there usually is not a one-to-one correspondence between the logical and development view.

4 Mainframe-Based Systems

Mainframe-based information systems are omni-present, and their limited flexibility is a problem to virtually any large organization in the world.

Many of these systems are essentially data-centered. Much of the system's functionality involves moving data from one place to another, such as reading data from user input screens, performing certain validation checks, and storing it in a central database, after which the data is processed and used to produce reports, mailings to customers, or as input for other on line screens. Typically, these systems have an on line mode, in which data typists or office personnel can enter or inspect data, and a batch mode, in which bulk processing to update data or produce reports is carried out, usually during the evening hours. The computational processing involved is often fairly simple. For architecture understanding purposes this means that an analysis of the data involved should play a key role. This, however, is usually complicated by the fact that many legacy systems do not use a relational database, as they rely on networked or hierarchical databases, or sometimes just plain flat files for storing data.

The most widely used programming language for these systems is Cobol. In a typical Cobol program, more than half of the code contains data structure declarations, defining how record structures are mapped onto memory. The other half consists of the procedural code, which also contains the system's business logic.

Due to the limited abstraction capabilities of Cobol, this business logic is difficult to isolate from the rest of the code. Within a Cobol program, all variables are visible in all procedures (called sections or paragraphs), and these procedures are not equipped with any parameter passing mechanism. A typical individual Cobol program is 1500 lines of code, but in some cases programs get as big as 30,000 lines of code.

The Cobol program acts as a module, and a Cobol system is built by letting Cobol programs call each other (using the CALL statement). With a program call, it is possible to pass parameters. Cobol systems may consist of as many programs as needed, but in practice such systems are divided into separate subsystems of approximately 500,000 lines of code, containing 200-500 programs. Cobol has no mechanisms for grouping programs into subsystems: naming conventions (letting every program start with a subsystem identifier) are typically used to make the subsystem partitioning visible.

Some form of code reuse is possible by using the Cobol copybook mechanism to include files. A copybook can contain any piece of text, and string replacements (crossing lexical boundaries) can be performed upon copybook inclusion. Copybooks are often used to define important data structures, such as records of database tables used throughout a system.

Batch processing is initiated by batch jobs, most commonly written in JCL. Batch jobs can start Cobol programs, and open, sort, or close files. Cobol programs that are called from batch jobs are referred to as main programs, whereas programs called by other Cobol programs are referred to as modules.

Comparing mainframe-based systems with Unix-based C applications two important differences show up. First, Cobol-based systems are much more data-centered. Second, the abstractions offered by Cobol are much more limited, having no parameter-passing procedure or function calling mechanism. On the other hand, Cobol has no pointer structures, thus simplifying data flow analysis (unfortunately, it is also complicated by the aliasing introduced by the REDEFINE construct, with which multiple record layouts can be given to a single memory area). Architectural analysis of mainframe based systems will have to take account of these differences. In practice, extraction techniques developed with C in mind (for example to support object identification for the purpose of migrating C-code to Java) are difficult to reuse in a Cobol setting [7].

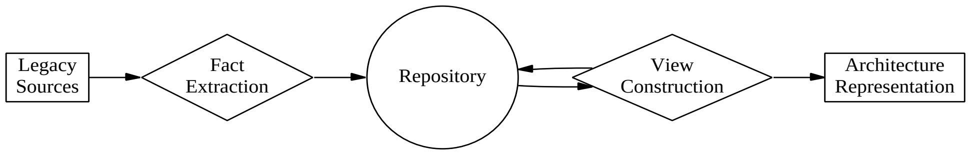

Figure 1: Architecture Extraction Tool Set Up

5 Tools for Architecture Understanding

Most existing architecture extraction tools (such as Rigi [29], Dali [14], the Software Bookshelf [13] and its portable companion [2]) work along the lines of model depicted in Figure 1. From the legacy sources, a set of facts is derived which are used to populate a repository. Example facts include call dependencies, copybooks used, tables used, and so on. Given the contents of the repository, a collection of views is constructed. Usually these views contain system hierarchy information indicating what the programs, databases, copybooks, and other components are, as well as various usage cross references. In addition to that, the views constructed can introduce new forms of abstraction, proposing architecture alternatives, such as new ways of modularization, or new ways of organizing a data model.

For the ideal architecture extraction tool set, we can identify a number of desirable characteristics concerning the way in which an extracted architecture is presented.

First, an architecture extraction tool does not try to extract *the* architecture, but rather helps the software engineer to build a mental model of a system's architectural structures.

Second, it helps to understand the relationships between these structures. In a Cobol system, for example, the modularization chosen, the calling structure of programs, and the use of database tables are three different structures that interact. Each of them can be listed as a separate view, using a system hierarchy, a call graph, and a CRUD (create, read, update, delete) matrix. Moreover, these views are related to each other, in that, for example, certain parts of the hierarchy may correspond to particular regions in the call graph, and that certain database tables may not used by these regions in the call graph. The extracted architecture makes it possible to explore such dependencies between the different views.

Third, an architecture extraction tool set permits zooming in from abstract to more specific information, or zooming out from particular data to a more abstract view. (called “reverse abstracting” in [25]). For example, if we look at module dependencies at a system level, we may detect anomalies, that is, unexpected dependencies. We then may wish to zoom in to the particular part of the call graph to see which particular CALL statement is responsible for this. Likewise, if we are about to change the interface of a given program, we may wish to zoom out to the call graph, in order to collect the set of programs affected by this change. Following the theory on program comprehension, we should permit *top-down* and *bottom up* understanding strategies, as well as the opportunistic combination between these [19, 22].

Fourth, architectures are presented as a combination of diagrams and text. Text used may include informal explanations, annotations made by software engineers inspecting the architecture, key comments extracted from the source, or selected pieces of system documentation. Text as well as diagrams can be used to list key metrics on module cohesion, coupling, or complexity. Boxes and and arrow diagrams are often used in architecture to present system elements and dependencies, such as call graphs, data flow, control flow, or entity-relationship diagrams. Diagrams and graphs are, whenever possible, extracted from the actual sources, so that they are consistent with the source at all times. Since such graphs may become very large, it is necessary to provide interactive zooming, filtering, and collapsing on extracted graphs. In fact, keeping these graphs manageable and understandable, by filtering out or abstracting from every irrelevant detail, is a key challenge to architecture extraction tools.

Fifth, the tools permits both browsing and searching a software architecture, as argued by [25]. Searching is planned activity with a specific goal, whereas browsing is more of an explorative strategy, with no fixed endpoint. Searching must be provided at an architectural level. Just opening up the repository for end-user querying is unlikely to provide a satisfactory solution, as this will require in-depth knowledge of the tool's underlying data model.

To facilitate all these presentation requirements, hypertext is a natural presentation candidate, as for example used by [4, 22, 6, 10]. Using standard browsers to inspect extracted architectures, helps to reduce the learning costs (as well as the installation costs for large organizations). Clickable graphs can be included using Adobe's PDF or W3C's XML-based scalable vector graphics (SVG) format. In fact, the combined extracted architectural structures can be viewed as a huge graph having typed edges (call, database usage, and other dependencies) between the nodes (programs, databases, copybooks, and so on). Hypertext makes it possible to navigate through this graph displaying a page for every node, and representing edges by hyperlinks.

Last but not least, architecture extraction tools must be able to recover an adequate number of different structures, enabling the software engineer to find answers to his architectural questions. In practice, a tool that is both rich in architectural structures, and that meets the various presentation requirements listed, does not exist yet.

6 Selected Extracted Structures

In existing, sizable systems, naming conventions are very important [20]. For Cobol systems, names of programs are architectural markers, and can be used to extract the system's development view. As Neighbours observes, in large systems, it is more important to determine the owning subsystem of a module rather than its function [20]. Names of Cobol programs (which can be eight character long) typically identify the system they belong to using the first two or three letters, as well as the sub (and sub-subsystem) they may belong to using some hierarchical numbering mechanism, and perhaps one or two extra letters to give a hint on the functionality.

An example illustrating the use of naming conventions is shown in Figure 2. This figure shows another structure, namely the sequence of screen handling programs in on line CICS application. Each ellipse is a program, and each arrow is a module activation originating from user interaction. All program names are 8 characters. The first two characters are always "RM" and indicate the system these modules belong to, (which performs Relation Management). The second two characters are numbers indicating the main screen sequences, such as RM01 (which may be used for "entering all data for a given customer"), RM03 ("change an address"), and so on. The remaining numbers indicate subsequences, such as RM010, RM011, RM012, etc. The last character indicates the sort of action performed by the program: an "S" is used for programs sending a screen, an "R" for receiving the screen contents, and an "I" for initiating a screen sequence. Comparing this naming convention with the figure, one can observe that it corresponds well to the logical structure of the program dependencies: calls to "S" programs, for example, are followed by "R" programs, and program sequences adhere to the numbering sequences.

A natural first step in understanding a legacy system's architecture is to explore the differences between the *development* view present in the program names used, and the *logical* view that can be extracted from actual statements in the source code. In the above example, Figure 2 displays that the naming conventions were indeed followed. Moreover, it shows that there are several independent sequences, for example in the top right and bottom left corner of the Figure. The most complicated dependencies are in the top left corner, where several major sequences interact.

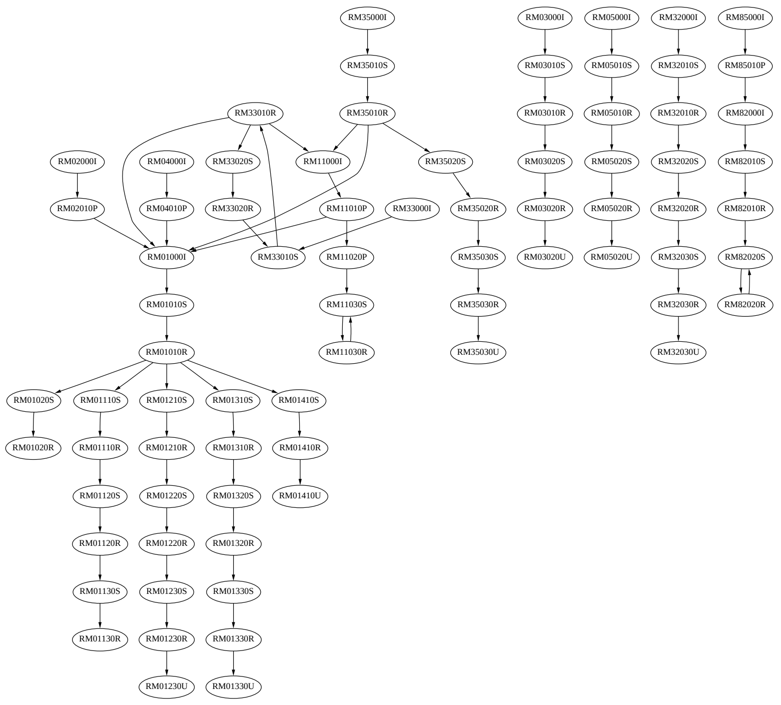

Figure 2: Screen sequence for an on line system

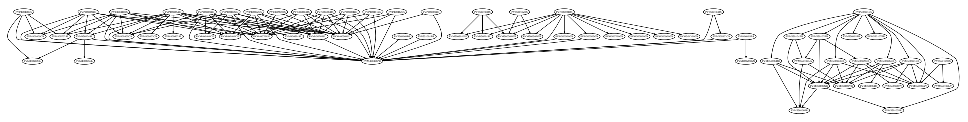

Figure 3: Example subsystem

In other examples, the correspondence between logical and development view may be more out of sync. As an example, we encountered a system consisting of 500,000 lines of code. The developers indicated that it was to be divided into 7 subsystems, which could easily be recognized from the program names. The subdivision was composed in such a way that there are no call dependencies between these seven subsystems, which makes them relatively independent. The call dependencies of one particular subsystem is shown in Figure 3. To our surprise, it contained one totally independent subgraph, at the right, which has no call dependencies to the other programs in the subsystem, indicating low cohesion of this subsystem.

One of the reasons that this separate cluster is part of the subsystem could be that there are other dependencies between these programs than just calling relationships. As an example, programs are called from batch jobs, and operate on files. These dependencies can be displayed as well, as shown in Figure 4. It displays how one particular batch job activates programs (boxes) that read (incoming arrow) or write (outgoing arrow) data files (ellipses), and how the batch job sorts (dashed arrow) or moves (dotted arrow) data files. Showing multiple dependencies in one graph, as in this figure, usually introduces so many additional edges that the graph becomes incomprehensible. Figure 4 still is comprehensible only because it focuses on just a single batch job.

At a larger scale, zooming out to *lifted* relations may help to keep the number of dependencies manageable [15]. For example, Figure 5 shows all database table dependencies

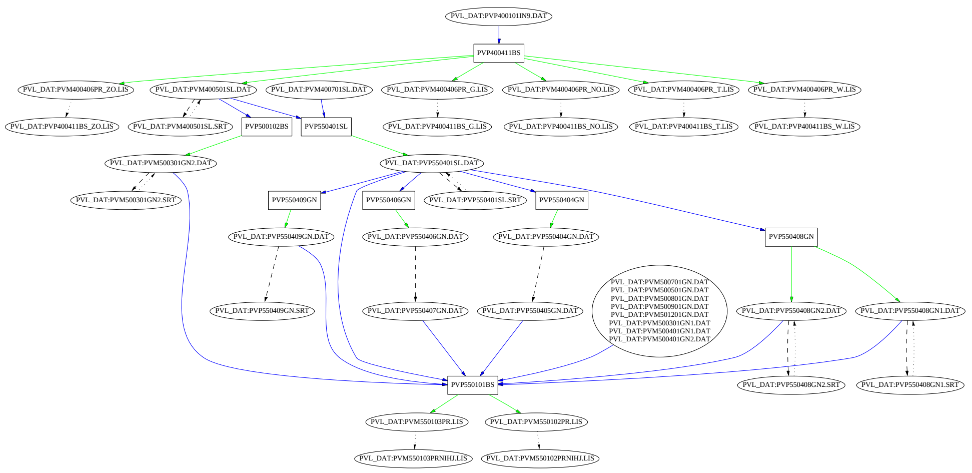

Figure 4: Batch job

between the seven different subsystems. Whenever a database table T is used in a program P in subsystem S, and T is also used in a program P' occurring in S', with $S \neq S'$, this dependency between programs P and P' is *lifted* to the system level, resulting in a dependency between S and S'. The figure shows the seven subsystems, and a line between two systems if they share a table dependency. The number indicates the cardinality of the unlifted relation, counting how many tables are jointly used by two systems. Thus, although the given decomposition present in the naming conventions yields a highly independent structure from the calling point of view, this is not the case for the database dependencies: there is not a single system that does not share a table with one of the other systems.

7 Interface Analysis

Software architecture is not only about components, but also about the interfaces between these components. What can we do to get a better understanding of the interfaces between the various legacy components?

Looking at component-based architectures such as EJB, COM, or Corba, we see that the type structure is a key mechanism to explain the interfaces between the classes: Classes export methods, and the signatures of these methods, defining the parameter types and the return type, comprise the class interface. The type helps to understand what set of values is permitted for a variable. Moreover, it allows to see when variables represent the same kind of entities. Third, it permits hiding the actual representation used (array versus record, length of array, ...), allowing a more abstract view of the variable.

Unfortunately, Cobol is an untyped language, seemingly blocking this route towards interface understanding. In this section, we will investigate how we can nevertheless *infer* types automatically from Cobol, for which we will use techniques covered in [8]. We will then see how these inferred types can help to acquire an architectural understanding of the interfaces of a legacy system. This illustrates how architecture extraction can go beyond mere fact collection and cross referencing: new levels of abstraction are "invented" to make the underlying system architecture explicit.

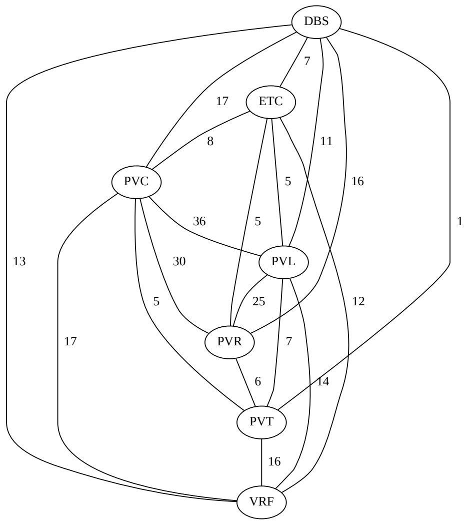

Figure 5: Database dependencies at the system level

7.1 Types for Cobol

In a typical Cobol system, explicit types could help to understand the system's architecture in the following ways:

- Types for the program's parameters describe the interface of that program;
- Types for on line screen inputs indicate what sort of data is entered and how it is used later on;
- Types for flat files describe the record structure;
- Types for records occurring in heavily used copybooks describe key data structures.

In addition to that, the *usage* of types throughout parts of the system could tell something about the functionality of particular parts: for example, if we are aware of a type *account*, and we see that it is heavily used in a particular set of programs, this can be a reason to group these programs into a module.

Unfortunately, Cobol is essentially an *untyped* language, without explicit types featuring the above advantages. In Cobol, a variable has to be declared before it is used, but this declaration only indicates how many bytes it occupies, whether these bytes are numbers or characters, and how variables are grouped into larger record-structured variables. Unfortunately, it is *not* possible to separate type definitions from variable declarations. This has three unpleasant consequences. First, when two variables need the same record structure, this structure is *repeated*. Second, whenever a data division contains a repeated record structure, the lack of type definitions makes it difficult to determine whether that repetition is accidental (the two variables are not related), or whether it is intentional (the two variables should represent the same sort of entity). Third, the absence of explicit types leads to a lack of abstraction, since there is no way to hide the actual representation of a variable into some type name.

7.2 Type Inference for Cobol Systems

To deal with the lack of types in Cobol, we will *infer* types for Cobol variables automatically by analyzing the *use* of these variables in the procedure division.

Primitive Types We distinguish three primitive types: (1) elementary types such as numeric values or strings; (2) arrays; and (3) records. Initially every declared variable gets a unique primitive type. Since (qualified) variable names must be unique in a Cobol program, they can be used as labels within a type to ensure uniqueness. We qualify these names with program or copybook names to obtain uniqueness at the system level. We use T_A to denote the primitive type of variable A.

Type Equivalence From *expressions* occurring in statements, an *equivalence relation* between primitive types is inferred. We distinguish three cases:

1. *Relational expressions* such as $v = u$ or $v \leq u$ result in an equivalence between T_v and T_u.
2. *Arithmetic expressions* such as $v + u$ or $v * u$ result in an equivalence between T_v and T_u.
3. *Array accesses* to the same array, such as $a[v]$ and $a[u]$ result in an equivalence between T_v and T_u.

We will generally speak of a *type*, meaning an *equivalence class of primitive types.* We will give names to types based on the names of the variables that are of that type. For example, the type of a variable with the name `L100-DESCRIPTION` will be called `DESCRIPTION`-type.

Subtyping From *assignment statements* a *subtype relation* between primitive types is inferred. From the assignment $v := u$ we conclude that T_u is *subtype* of T_v, i.e., v can hold at least all the values u can hold.

Union types From Cobol *redefine clauses*, a *union type* relation between primitive types is inferred. When an entry v in the data division redefines an entry u, we conclude that T_v and T_u are part of the same *union type*.

System-Level Analysis The type relations described before are derived at the program level. We also derive a number of type relations at the system-wide level: (1) *program parameters:* the types of the actual parameters of a program call (listed in the Cobol `USING` clause) are *subtypes* of the formal parameters (listed in the Cobol `LINKAGE` section), (2) *file/table access:* variables read from or written to the same file or table have *equivalent* types, and (3) *copybooks:* a variable which is declared in a copybook gets the same type in all the programs that include this copybook.

Literals Whenever a literal value l is assigned to, or compared with a variable v, we infer that l is a *permitted value* for the type of v. If additional analysis indicates that variables in this type are only assigned values from this set of literals, we can infer that the type in question is an *enumeration type*.

Aggregate Structure Identification Whenever the types of two records are related to each other, types for the individual fields should be propagated as well. If the two record structures are identical (having the same number of fields, each of the same size) each individual field type can be propagated. Eidorff *et al.* [11] and Ramalingam *et al.* [23] have published an algorithm which splits aggregate structures in smaller “atoms”, such that types can be propagated through record fields even if the records do not have the same structure.

7.3 Case Study

We have applied this type inferencing approach to a Cobol/CICS legacy system called *Mortgage*, comprising approximately 100,000 lines of code. It consists of an on-line (interactive) part, as well as a batch part, and it is in fact a subsystem of a larger (1 MLOC) system.

To analyze the architecture of *Mortgage* we have used a tool for type-based browsing of legacy systems called TYPEEXPLORER. This tool aims at presenting a legacy system according to the presentation principles discussed in Section 5, giving inferred types a prominent role. TYPEEXPLORER is still in development, and is described in more detail in [9].

A type-based exploration of *Mortgage*'s architecture starts with a list of all programs together with their inferred signatures, which describe the types of the program parameters. For our *Mortgage* system, an immediate observation is that the type of the first formal parameter of all batch programs is the same – the *program-fields* type. This raises the question why this is so, and what sort of type this *program-fields* type is. Inspection shows us that it is a record-type, storing the name of the program, the current status, the name of the files currently processed, etc. Moreover, it holds data which is not necessary for the proper execution of the program. Instead, the data is used to quickly find the program responsible for the problems if one of the batch runs crashes.

This shared first parameter shown by TYPEEXPLORER thus immediately leads to an architectural requirement, namely that the system should support fast repairs and restarts at the proper position whenever one of the batch runs crashes in the middle of the night.

The inferred types also show us that this convention is actually used. The *program-fields* record contains one field (the *subroutine* field) holding the name of the program currently being run. Type inferencing collects all literal values that are used for (i.e., assigned to variables of) the type *subroutine*. This list exactly corresponds to the list of all batch programs, which is the result of the fact that each program correctly starts by setting the *subroutine* field to the program's name.

It is interesting to observe that *Mortgage* also clearly shows that just looking at the *names* of formal parameters is not sufficient. To see why this is so, we take a look at the *on line* part of *Mortgage* (the part invoked from screens via CICS) . The first parameter of each on line programs is the same, namely `DFHCOMMAREA`. However, they all have a different type! All `DFHCOMMAREA` variables are strings of different lengths. The specific name `DFHCOMMAREA` is required by CICS. The first thing each program does is to assign that variable to a more structured record variable. It is the type of that structured record variable that TYPEEXPLORER recognizes as the appropriate type for the first parameter of the linkage sections, which it displays in the inferred signature.

Type inferencing also helps to understand the meaning of the program parameters. For example, many programs in *Mortgage* have integer-valued numbers as parameters (having picture string `S(9) COMP-3`). Often, these are in fact enumeration types, in which case TYPEEXPLORER recognizes them as such. Several programs turn out to have a parameter named *function*, with 5 to 10 permitted values. Based on this function value, the program performs one of several functions. This leads us to two design decisions: different (but related) functions are grouped into programs, and the mechanism used is a switch on an enumerated value, instead of the Cobol feature in which one program can have multiple entry points.

Last but not least, TYPEEXPLORER shows how such *function* enumeration parameters are passed from one program to another. As an example, one of the *Mortgage* programs contains a parameter for determining how a person's name is formatted (full first names, one initial only, with title, and so on), and another to format street names (capitalized, street abbreviated, and so on). One of the top level programs has 10 different parameters, corresponding to these formatting codes. The types inferred exactly show how each of the codes (which are all integer numbers) correspond to the parameters of the various formatting

	P_1	P_2	P_3	P_4
NAME	√			
TITLE	√			
INITIAL	√			
PREFIX	√			
NUMBER				√
NUMBER-EXT				√
ZIPCD				√
STREET			√	√
CITY		√		√

Table 1: Example Feature Table

programs.

In short, type-based browsing can be used to discuss whether requirements such as crash recovery are properly supported, how functionality is grouped in modules, and how modules are dependent via types. Other architectural issues can be identified as well, by studying the type relationships between copybooks, the use of database record types across programs, and so on.

8 Renewing Architectures

A more ambitious form of architectural understanding is to look at the data and functionality of a legacy system in a fresh way, independent of the current development view. One possibility is to come up with a potentially completely new system modularization. An example is to take an object-oriented point of view on a given legacy system. This requires an analysis of data fields (which may become attributes), functionality manipulating these fields (which may become methods), and suitable combinations of these, (thus arriving at classes). This then paves the way for architecture renewal, where existing legacy data and functionality are reorganized in order to make them accessible via a more flexible architecture.

In this section, we will look at object identification, that is, viewing existing systems in an object-oriented manner. Several techniques for distilling object structures from legacy systems exist, covered, for example, in [21, 27, 12]. Most of these are based on system remodularization using some form of *cluster analysis*, of which a survey is provided by Lakhotia [17]. In this section, we will work with another technique called *concept analysis*, which recently has been proposed as a tool for analyzing the modular structure of legacy code [18, 24]. We will essentially follow [7], which also contains a detailed comparison between cluster and concept analysis.

8.1 Concept Analysis

Concept analysis starts with a set $\mathcal{M}$ of *items*, a set $\mathcal{F}$ of *features*,[1] and a *feature table* (relation) $T \subseteq \mathcal{M} \times \mathcal{F}$ indicating the features possessed by each item. An example feature table is shown in Table 1. It lists the names of 9 example Cobol record fields, and their use in 4 different programs $P_1 \dots P_4$.

For a set of items $I \subseteq \mathcal{M}$, we can identify the *common features*, written $\sigma(I)$, via:

$$\sigma(I) = \{ f \in \mathcal{F} \mid \forall i \in I : (i, f) \in T \}$$

For example, $\sigma(\{\texttt{ZIPCD}, \texttt{STREET}\}) = \{P_4\}$. Likewise, we define for $F \subseteq \mathcal{F}$ the set of

[1]The literature generally uses *object* for *item*, and *attribute* for *feature*. In order to avoid confusion with the objects and attributes from object orientation we have changed these names into items and features.

name	extent	intent
top	{NAME, TITLE, INITIAL, PREFIX, NUMBER, NUMBER-EXT, ZIPCD, STREET, CITY}	$\emptyset$
c1	{NAME, TITLE, INITIAL, PREFIX}	$\{P_1\}$
c2	{NUMBER, NUMBER-EXT, ZIPCD, STREET, CITY }	$\{P_4\}$
c3	{STREET}	$\{P_3, P_4\}$
c4	{CITY}	$\{P_2, P_4\}$
bot	$\emptyset$	$\{P_1, P_2, P_3, P_4\}$

Table 2: All concepts in the example of Table 1

common items, written $\tau(F)$, as:

$$\tau(F) = \{i \in M \mid \forall f \in F : (i, f) \in T\}$$

For example, $\tau(\{P_3,P_4\}) = \{\texttt{STREET}\}$.

A *concept* is a pair (I,F) of items and features such that $F = \sigma(I)$ and $I = \tau(F)$. In other words, a concept is a maximal collection of items sharing common features. In our example,

$$(\{\texttt{NAME},\texttt{TITLE},\texttt{INITIAL},\texttt{PREFIX}\},\{P_1\})$$

is the concept of those items having feature P_1, i.e., the fields used in program P_1. All concepts that can be identified from Table 1. are summarized in Table 2. The items of a concept are called its *extent*, and the features its *intent*.

The concepts of a given table form a partial order via:

$$(I_1,F_1) \leq (I_2,F_2) \;\Leftrightarrow\; I_1 \subseteq I_2 \;\Leftrightarrow\; F_2 \subseteq F_1$$

As an example, for the concepts listed in Table 2, we see that bot $\leq c3 \leq c2 \leq$ top.

The subconcept relationship allows us to organize all concepts in a *concept lattice*, with *meet* $\wedge$ and *join* $\vee$ defined as

$$\begin{aligned}(I_1,F_1) \wedge (I_2,F_2) &= (I_1 \cap I_2, \sigma(I_1 \cap I_2)) \\ (I_1,F_1) \vee (I_2,F_2) &= (\tau(F_1 \cap F_2), F_1 \cap F_2)\end{aligned}$$

The visualization of the concept lattice shows all concepts, as well as the subconcept relationships between them. For our example, the lattice is shown in Figure 6. In such visualizations, the nodes only show the “new” items and features per concept. More formally, a node is labelled with an item i if that node is the *smallest* concept with i in its extent, and it is labelled with a feature f if it is the *largest* concept with f in its intent.

The concept lattice can be efficiently computed from the feature table; we refer to [18, 24, 28] for more details.

8.2 Case Study

We have performed several experiments with the use of concept analysis for the *Mortgage* case study also used in Section 7. We tried to extract an object-oriented structure by reorganizing database fields according to their use in programs.

When applying any remodularization technique, it is essential to restrict the input to relevant data only. For example, some programs may be technical in nature, dealing with error handling or logging. Other programs may be control modules, not containing any interesting functionality. Filtering out such programs, for example based on fan-in / fan-out metrics, is necessary in order to arrive at meaningful cluster or concept analysis results. For our case study, we used selection criteria discussed in [5, 7]: the items are the fields of persistent data records, and the programs are restricted to those with a low fan-in and fan-out.

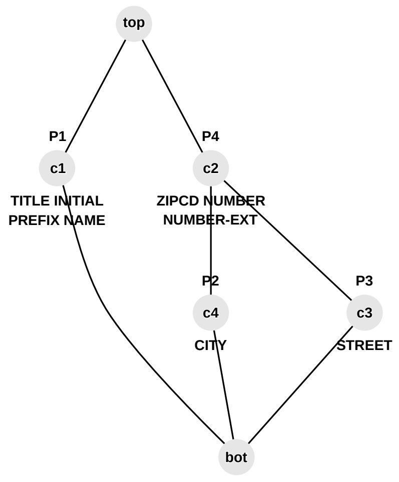

Figure 6: Lattice for the concepts of Table 2

The resulting concept lattice is shown in Figure 7. Each node represents a concept. The items (field names) are names written below the concept, the features (programs using the fields) are written as numbers above the concept. Exploring the lattice provides insight in the organization of the *Mortgage* legacy system, and gives suggestions for grouping programs and fields into classes.

The row just above the bottom element consists of five separate concepts, each containing a single field. As an example, the leftmost concept deals with *mortgage numbers* stored in the field `MORTGNR`. With it is associated program `19C`, which according to the comment lines at the beginning of this program performs certain checks on the validity of mortgage numbers. This program *only* uses the field `MORTGNR`, and no other ones.

As another example, the concept `STREET` (at the bottom right) has three different programs directly associated with it. Of these, `40` and `40C` compute a certain standardized extract from a street, while program `38` takes care of standardizing street names.

If we move up in the lattice, the concepts become larger, i.e., contain more items. The leftmost concept at the second row contains *three* different fields: the *mortgage sequence number* `MORTSEQNR` written directly at the node, as well as the two fields from the lower concepts connected to it, `MORTGNR` and `RELNR`. Program `09` uses all three fields to search for full mortgage and relation records.

Another concept of interest is the last one of the second row. It represents the combination of the fields `ZIPCD` (zip code), `HOUSE` (house number), and `CITYCD` (city code), together with `STREET` and `CITY`. This combination of five is a separate concept, because it actually occurs in four different programs (`89C`, `89`, `31C`, `31`). However, there are no programs that *only* use these variables, and hence this concept has no program associated with it.

The largest concepts reside in the top of the lattice, as these collect all fields of the connected concepts lower in the lattice. For example, the concept with programs `31` and `31C` consists of a range of fields directly attached with it (`FIRSTNM`, ...), as well as of all those in the three downward links below it. It corresponds to almost all fields of one particularly large record, holding the data of so-called *relations* (people and companies that play a role when a mortgage is set up). These fields are then processed by programs `31` and `31C`. Only one field, `MOD-DAT` (modification date), is part of that *relations* record but

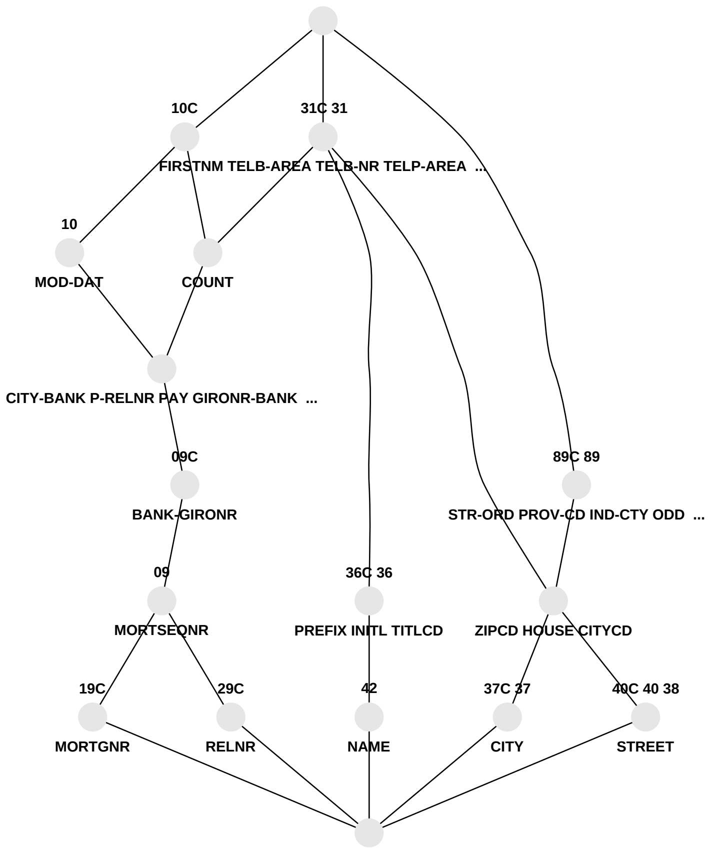

Figure 7: Concept lattice showing how persistent fields are used in programs in the *Mortgage* case study.

not used in `31` and `31C`.

Another large concept of interest is the one with programs `89C` and `89`. The fields in this concept all come from the Dutch *zip code book*, holding data for all Dutch addresses and their zip codes. As can be seen from Figure 7, the fields of this concept are largely disjoint with those of the *relations* concept (with programs `31` and `31C`). However, these two concepts also share five fields, namely those of the `ZIPCD` concept. These fields can be used (in various combinations) as the lookup key for the zip code book.

In short, each concept is a candidate class. The smallest concept introducing a field corresponds to the class having that field as attribute. The largest concept with a given program as feature corresponds to the class with that program attached as method to it. This is reflected in the way the concepts are labeled with items and features in the concept lattice. Classes close to the bottom are the smallest classes (containing few attributes).

The subconcept relationship corresponds to class relations. Typically, a class for a concept c is composed via aggregation from the classes of the subconcepts of c. Alternatively, if a concept c has a subconcept c', c may be composed from c' via inheritance. As an example, the concept with field `NAME` (and program `42`) in Figure 7 deals with names of persons. A natural refinement of this class is the concept above it, which extends a person's name with his prefixes, initials, and title code. Independent "columns" in the concept lattice correspond to separate class hierarchies.

9 Concluding Remarks

In this paper, we have looked at architecture from the perspective of legacy systems. We addressed the following issues:

1. Legacy systems hardly ever have a documented architecture: if they do, it is likely to be out of date.
2. Having a documented architecture available helps in dealing with legacy systems, for example when companies merge, when new software engineers are added to a maintenance team, or when legacy functionality needs to be made available through new channels.
3. Extracting architecture from legacy systems involves finding architectural *structures*.
4. Mainframe-based legacy systems pose particular architectural extraction requirements, such as the need to take a data-centered point of view.
5. In order to present the extracted architectural structures, graph visualization, zooming in and out on abstractions, and the ability to see the relationships between extracted structures are essential. Hypertext-based presentations allow the software engineer to navigate easily through an extracted architecture.
6. Decompositions based on naming conventions are a natural starting point for architecture browsing: contrasting this development view with the logical view extracted from the source code statements yields the possibility to assess the quality of the decomposition used.
7. Types are an often-used mechanism to describe component interfaces; Even if the underlying legacy language (such as Cobol) is untyped, types can be inferred in order to describe legacy component interfaces.
8. Decompositions based on system structure can be identified using concept and cluster analysis. This can even lead to new views on a legacy system, in which the original functional decomposition is replaced by an object-oriented decomposition.

We elaborated on each of these issues in a separate section, including references to key publications. We discussed practical experience with the use of architectural understanding tools and techniques. We encountered many different architectural structures, such as screen sequences, call relationships, class proposals, database usage, inferred types, and so on, and we illustrated how these can be distilled from actual legacy systems (mostly business data processing systems written in Cobol).

References

[1] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[2] I. T. Bowman, R. C. Holt, and N. V. Brewster. Linux as a case study: Its extracted software architecture. In *21st International Conference on Software Engineering, ICSE-99*, pages 555–563. ACM, 1999.

[3] M. L. Brodie and M. Stonebraker. *Migrating Legacy Systems: Gateways, interfaces and the incremental approach*. Morgan Kaufman Publishers, 1995.

[4] P. Brown. Integrated hypertext and program understanding tools. *IBM Systems Journal*, 30(3):363–392, 1991.

[5] A. van Deursen and T. Kuipers. Rapid system understanding: Two COBOL case studies. In *Sixth International Workshop on Program Comprehension; IWPC'98*, pages 90–98. IEEE Computer Society, 1998.

[6] A. van Deursen and T. Kuipers. Building documentation generators. In *International Conference on Software Maintenance, ICSM'99*, pages 40–49. IEEE Computer Society, 1999.

[7] A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *21st International Conference on Software Engineering, ICSE-99*, pages 246–255. ACM, 1999.

[8] A. van Deursen and L. Moonen. Type inference for COBOL systems. In *Proceedings of the fifth Working Conference on Reverse Engineering, WCRE'98*, pages 220–230. IEEE Computer Society, 1998.

[9] A. van Deursen and L. Moonen. Exploring legacy systems using types. In *Proceedings of the 7th Working Conference on Reverse Engineering, WCRE'2000*. IEEE Computer Society, 2000. To appear.

[10] Automatic documentation generation; white paper. Software Improvement Group, 2001. `http://www.software-improvers.com`.

[11] P. H. Eidorff, F. Henglein, C. Mossin, H. Niss, M. H. Sorensen, and M. Tofte. Anno Domini: From type theory to Year 2000 conversion tool. In *26th Symp. on Principles of Progr. Languages, POPL'99*. ACM, 1999.

[12] H. Fergen, P. Reichelt, and K. P. Schmidt. Bringing objects into COBOL: MOORE - a tool for migration from COBOL85 to object-oriented COBOL. In *Proceedings of the Conference on Technology of Object-Oriented Languages and Systems (TOOLS 14)*, pages 435–448. Prentice-Hall, 1994.

[13] P. J. Finnigan, R. C. Holt, I. Kalas, S. Kerr, K.Kontogiannis, H. A. Müller, J. Mylopoulos, and S. G. Perelgut. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, 1997.

[14] R. Kazman and J. Carrière. Playing detective: Reconstructing software architecture from available evidence. *Automated Software Engineering*, 6:107–138, 1999.

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

[16] P. B. Kruchten. The 4 + 1 view model of architecture. *IEEE Software*, pages 42–50, November 1995.

[17] A. Lakhotia. A unified framework for expressing software subsystem classification techniques. *Journal of Systems and Software*, pages 211–231, March 1997.

[18] C. Lindig and G. Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *19th International Conference on Software Engineering, ICSE-19*, pages 349–359. ACM Press, 1997.

[19] A. von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *IEEE Computer*, pages 44–55, August 1995.

[20] J. M. Neighbors. Finding reusable software components in large systems. In *3rd Working Conference on Reverse Engineering; WCRE'96*, pages 2–10. IEEE Computer Society, 1996.

[21] P. Newcomb and G. Kottik. Reengineering procedural into object-oriented systems. In *Second Working Conference on Reverse Engineering; WCRE'95*, pages 237–249. IEEE Computer Society, 1995.

[22] V. Rajlich and S. Varadarajan. Using the web for software annotations. *Int. Journal of Software Engineering and Knowledge Engineering*, 9(1):55–72, 1999.

[23] G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *26th Symp. on Principles of Progr. Languages, POPL'99*. ACM, 1999.

[24] M. Siff and T. Reps. Identifying modules via concept analysis. In *International Conference on Software Maintenance, ICSM97*. IEEE Computer Society, 1997.

[25] S. E. Sim, C. L. A. Clarke, R. C. Holt, and A. M. Cox. Browsing and searching software architectures. In *Int. Conf. on Software Maintenance, ICSM'99*, pages 381–390. IEEE Computer Society, 1999.

[26] S. E. Sim and R. C. Holt. The ramp-up problem in software projects: A case study of how software immigrants naturalize. In *20th Int. Conf. on Software Engineering; ICSE-97*, pages 361–370. ACM, 1998.

[27] H. M. Sneed and E. Nyáry. Extracting object-oriented specification from procedurally oriented programs. In *Second Working Conference on Reverse Engineering; WCRE'95*, pages 217–226. IEEE Computer Society, 1995.

[28] G. Snelting. Concept analysis — a new framework for program understanding. In *Proceedings of the ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)*, 1998. SIGPLAN Notices 33(7).

[29] K. Wong, S.R. Tilley, H.A. Müller, and M.-A.D. Storey. Structural redocumentation: a case study. *IEEE Software*, 12(1):46–54, 1995.