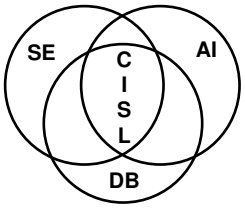
TR-0222-10-92-165

DARWIN: On the Incremental Migration of Legacy Information Systems[1]

Michael L. Brodie
GTE Laboratories, Inc.
40 Sylvan Road
Waltham, MA 02254

Michael Stonebraker
College of Engineering
University of California, Berkeley
Berkeley, CA 94720

March 1993



---

1 This report is the result of the DARWIN project initiated by Michael Stonebraker in 1991. It is also available as a Technical Memorandum of Electronics Research Laboratory, College of Engineering, University of California, Berkeley.

This is an advanced version of a monograph to be published by Morgan Kaufmann Publishers in the first quarter of 1994.

Abstract

As is painfully evident today, the deterioration of the transportation, education[2], and other national infrastructures negatively impacts many aspects of life, business, and our economy. This has resulted, in part, when responses to short term crises discourage investing in infrastructure enhancement and when there are no effective means for infrastructure evolution. This paper responds to the deterioration of the information system (IS) infrastructure that has strong negative impacts on ISs, on the organizations they support, and, ultimately, on the economy. This paper addresses the problem of legacy IS migration by methods that mediate between short term crises and long term goals. It presents an effective strategy and a spectrum of supporting methods for migrating legacy ISs into a target environment that includes *rightsized* hardware and modern technologies (i.e., infrastructure) such as a client-server architecture, DBMSs and CASE. We illustrate the methods with two migration case studies of multi-million dollar, mission critical legacy ISs. The contribution of this paper is a highly flexible set of migration methods that is tailorable to most legacy ISs and business contexts. The goal is to support continuous, iterative evolution. The critical success factor, and challenge in deployment, is to identify appropriate portions of the IS and the associated planning and management to achieve an incremental migration that is feasible with respect to the technical and business requirements. The paper concludes with a list of desirable migration tools for which basic research is required. The principles described in this paper can be used to design future ISs and an infrastructure that will support continuous IS evolution to avoid future legacy ISs.

---

2 The Edison project of Whittle Communications L.P., is attempting to create a innovative, for-profit alternative to the government funded school systems. It is based on the principle that Darwinian evolution applies not only to the species, but also to our institutions and organizations.

TABLE OF CONTENTS

1. MIGRATING LEGACY INFORMATION SYSTEMS 1
   - 1.1 Strategies 2
   - 1.2 Architectures 4
   - 1.3 Methods 7
2. MIGRATING DECOMPOSABLE LEGACY ISs 12
   - 2.1 Forward Migration Method For Decomposable Legacy ISs 12
   - 2.2 Reverse Migration Method For Decomposable Legacy ISs 17
   - 2.3 General Migration Method For Decomposable Legacy ISs 19
3. CASE STUDY 1 MIGRATING CMS 24
   - 3.1 CMS 24
   - 3.2 Analysis of the CMS Core 25
   - 3.3 The CMS Migration Plan 26
   - 3.4 CMS Cut-over Issues 30
   - 3.5 CMS Migration Summary 31
4. MIGRATING SEMI-DECOMPOSABLE LEGACY ISs 32
   - 4.1 Migration Method For Semi-decomposable Legacy ISs 32
5. CASE STUDY 2 MIGRATING TPS 34
   - 5.1 TPS 34
   - 5.2 TPS Analysis and Migration Challenges 36
   - 5.3 The TPS Migration Plan, Part I 38
6. MIGRATING NON-DECOMPOSABLE LEGACY ISs 43
   - 6.1 Migration Method For Non-decomposable Legacy ISs 43
7. MIGRATING LEGACY ISs (GENERAL CASE) 45
   - 7.1 The TPS Migration Plan, Part II 45
8. RESEARCH AGENDA 48
   - 8.1 Gateways 48
   - 8.2 Specification Extractor 48
   - 8.3 Application Analyzer 48
   - 8.4 Dependency Analyzer 49
   - 8.5 Migration Schema Design and Development Tools 49
   - 8.6 Database Extractor 49
   - 8.7 Higher Level 4GL 49
   - 8.8 Performance Tester 50
   - 8.9 Application Cut-over 50
   - 8.10 Distributed IS Development and Migration Environment 50
   - 8.11 Migration Planning and Management 50
   - 8.12 Distributed Computing 51
9. CONCLUSIONS AND EPILOGUE 52

ACKNOWLEDGEMENTS 54

REFERENCES 55

1. MIGRATING LEGACY INFORMATION SYSTEMS

Most large organizations are deeply mired in their information systems (IS) *sins of the past*. Typically, their ISs are large (e.g., 10^7 lines of code), geriatric (e.g., more than 10 years old), written in COBOL, and use a legacy database service (e.g., IBM's IMS or no database management system (DBMS) at all). These ISs are mission critical (i.e., essential to the organization's business) and must be operational at all times. These characteristics define what we call **legacy information systems (legacy IS)**. Today, legacy ISs pose one of the most serious problems for large organizations. Costs due to problems of a single legacy IS (e.g., failures, maintenance, inappropriate functionality, lack of documentation, poor performance) can often exceed hundreds of millions of dollars per year. They are not only inordinately expensive to maintain, but also *inflexible* (i.e., difficult to adapt to changing business needs), and *brittle* (i.e., easily broken when modified for any purpose). Perhaps worse is the widespread fear that legacy ISs will, one day, break beyond repair. Such fears combined with a lack of techniques or technology to fix legacy IS problems result in ***IS apoplexy.*** That is, legacy ISs consume 90% to 95% of all IS resources. This prevents organizations from moving to newer software, such as client-server configurations, current generation DBMSs, and fourth generation languages (4GLs). Consequently, organizations are prevented from *rightsizing* that involves moving from large mainframe computers to smaller, less expensive computers that fully meet current IS requirements. This apoplexy, in turn, is a key contributor to the software crisis. New requirements, often called the IS backlog, cannot be met since legacy ISs cannot be extended and new ISs cannot be developed with the 5% to 10% remaining resources. These problems are both key motivations of and major roadblocks to the world-wide movement to re-engineer corporations and their major ISs.

In legacy IS migration, an existing IS is evolved into a target IS by replacing the hardware and much of the software including the interfaces, applications, and databases. Under some circumstances, some existing components can, and should, be incorporated into the target environment. In this paper, the ***target environments*** are intended to take maximum advantage of the benefits of *rightsized* computers, a client-server architecture, and modern software such as relational DBMSs, 4GLs, and CASE. For example, a modern DBMS (e.g., backup, recovery, transaction support, increased data independence, performance improvements) assists in increasing control over the IS (e.g., maintenance). It also provides a basis for future evolution and integration with other ISs. For example, a DBMS could facilitate *data liberation* (i.e., any application could use the DBMS to access valuable corporate data that is currently inaccessible due to the legacy database service).

A fundamental goal of legacy IS migration is that the target IS not become a legacy IS. When the migration is complete, the target IS is fully operational in the target environment. All application code and user interfaces are completely written in a 4GL and the database resides on a current generation DBMS. A wise choice of target environment will facilitate the application being moved to a wide variety of current and future desktop machines and the database being deployed on a wide variety of computing platforms. Hence, the target IS can readily be ported to environments appropriate to current and future requirements. That is, the target IS is designed to be very flexible (e.g., portable) for current and future *rightsizing* and to avoid becoming a future legacy IS.

In the rest of the introduction, we discuss migration strategies and select one, list migration objectives, and introduce the alternative migration architectures and methods that are presented in the paper.

1.1 Strategies

We now describe the motivating problems and the key to their solution in the context of two strategies for migrating legacy ISs, **Cold Turkey** and **Chicken Little**.

Cold Turkey involves rewriting a legacy IS from scratch to produce the target IS using modern software techniques and hardware of the target environment. This strategy carries substantial risk of failure for the following reasons.

• A better system must be promised.

It is nearly impossible to propose a *one-for-one* rewrite of a complex IS. Management will rarely budget the required major expenditure if the only payoff is to lower future maintenance costs. Additional business functions must be promised. This adds complexity to the replacement IS and increases the risk of failure.

• Business conditions never stand still.

The development of large, complex ISs requires years to accomplish. While the legacy IS rewrite proceeds, the original legacy IS evolves in response to maintenance and urgent business requirements, and by *midnight functions* (i.e., features installed by programmers in their spare time). It is a significant problem to evolve the developing replacement IS in step with the evolving legacy IS.

More significant than maintenance and minor *ad hoc* changes are changes in the business processes that the IS is intended to support. These are typically in a constant state of flux. The prospect of incorporating support for new business processes in the replacement IS may lead to significant changes to the IS's requirements throughout its development. This also increases the risk of failure.

• Specifications rarely exist.

The only documentation for legacy ISs is typically the code itself. The original implementors have long since departed. Documentation is often non-existent, out of date, or has been lost. The original specifications and coding practices are now considered primitive or bad (e.g., self-modifying code). For example, the code is often the only documentation for the commonplace *variant record* encodings in which the interpretation of one data element is controlled by another data element. Often, legacy code was written for high performance on some extinct computer, resulting in arcane code constructs.

In such situations, the exact function of the legacy IS must be *decrypted* from the code, if it is to be understood or copied in the replacement IS. This adds greatly to the complexity and cost of developing the replacement IS.

• Undocumented dependencies frequently exist.

Invariably, applications, from non-critical (e.g., reporting programs) to mission critical, access the legacy IS for its mission critical information and other resources. Over the ten plus year life of the legacy IS, the number of these dependent ISs grows (e.g., 1,200 in a case study described below), few of which may be known to the legacy IS owners. The process of rewriting legacy ISs from scratch must identify and accommodate these dependencies. This again adds to the complexity of the rewrite and raises the risk of failure of dependent ISs.

• Legacy ISs can be too big to cut-over.

Many legacy ISs must be operational almost 100% of the time. Many legacy databases or files require weeks to dump or download. Even if the rewritten IS were fully operational, there are no techniques to migrate the live data from the legacy IS to the new IS within the time that the business can support being without its mission critical IS. Live data must also be converted to fit the new system, again increasing project time and complexity. This may not just add complexity, it often prohibits Cold Turkey altogether.

• Management of large projects is hard.

The difficulty of most large projects is seriously under-estimated. Hence, there is a tendency for them to grow uncontrollably in head count. Few organizations are capable of managing the development of an IS with the several hundred contributors that are common for ISs of the size and complexity we are considering. Managing more and more people inevitably brings on the famous *Brooks effect* [BROO75] resulting in less and less useful work.

• Lateness is seldom tolerated.

Large projects are inevitably late due to the problems cited above. Management patience wears out quickly, especially in organizations whose basic function is not software production. This frequently results in the termination of partly or mostly completed projects.

• Large projects tend to bloat.

There is a tendency for large projects to become bloated with nonessential groups. For example, for a project as critical as a legacy IS migration, organizations may want to explore the introduction of new management techniques and technologies (e.g., Re-engineering, CASE). This is often done by adding additional groups to the already large project. Groups that are not critical to the migration itself increase the budget and management complexity, thus making the project more vulnerable to termination.

Cold Turkey involves high risk. It has been applied and has failed many times in large organizations. We now turn our attention to the alternative, low-risk and novel strategy, **Chicken Little**, the focus and contribution of this paper.

Chicken Little involves migrating the legacy IS, in place, by small incremental steps until the desired long term objective is reached. Each step requires a relatively small resource allocation (e.g., a few person years), a short time, and produces a specific, small result towards the desired goal. This is in sharp contrast to the vast resource requirements of a complete rewrite (e.g., hundreds of person years), a multi-year development, and one massive result. If a Chicken Little step fails, only the failed step must be repeated rather than the entire project. Since steps are designed to be relatively inexpensive, such incremental steps do not need to promise dramatic new function to get funded.

Each problem cited in Section 1.1 can be addressed in an incremental fashion. In addition, failures in individual steps may indicate large or previously unforeseen problems. Due to the incremental nature of Chicken Little, such problems can be addressed incrementally. Hence, Chicken Little is much safer and more feasible than Cold Turkey.

In this paper, we investigate and apply the Chicken Little migration strategy to legacy ISs, from well structured to unstructured. The key to successful Chicken Little migration, and

its principal challenge, concerns the selection of independent increments to migrate (i.e., portions of legacy interfaces, applications, and databases that can be migrated independently of each other), the sequencing of the increments to achieve the desired goal, and dealing with unavoidable problems (e.g., dependencies between migration steps).

1.2 Architectures

All ISs can be considered as having three functions: interfaces, applications, and a database service. The architectures of legacy ISs vary widely on a spectrum from well-structured (e.g., modular, hence decomposable) to unstructured[3] (e.g., non-decomposable).

The best architecture for migration purposes is a decomposable structure in which the interface, application, and database services can be considered as distinct components with well-defined interfaces. Figure 1.1 illustrates a **decomposable legacy IS** that consists of a collection of application modules (M_i) each interacting with a database service and each, potentially, with its own user interface (UI_j) and system level interface (SI_j) through which it interacts with one or more IS. Interfaces, both user and system level, must be considered separately since they differ significantly in technology, design, performance requirements, and impact (e.g., number and requirements of human users versus ISs accessing the legacy IS). The prime requirements for an architecture to be decomposable are that the application modules are independent of each other (e.g., have no hierarchical structure) and interact only with the database service.

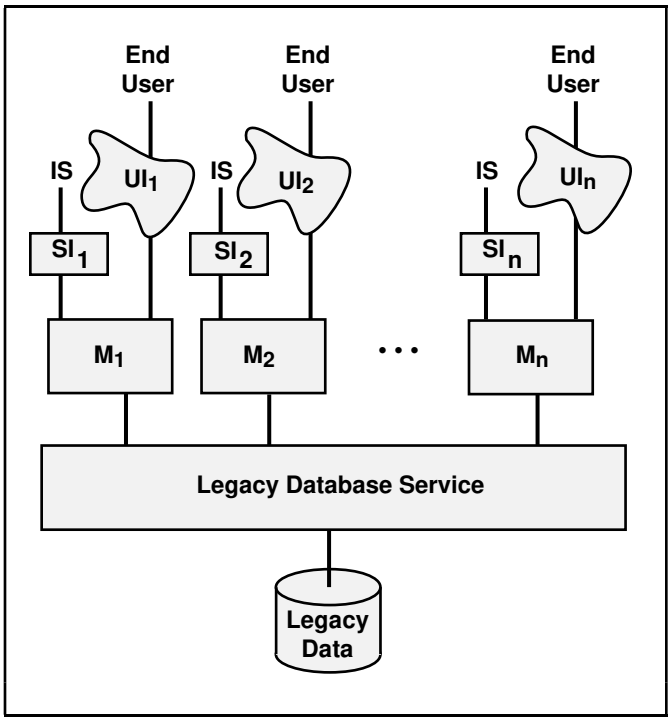


Figure 1.1: Decomposable Legacy IS Architecture

[3] The reasons for a lack of structure are many. They relate to the state of the technology at the time of design, the design and development methods and tools, uncontrolled enhancements and maintenance, lack of documentation, and to many other aspects of ISs.

A worse architecture for migration purposes is a **semi-decomposable legacy IS**, illustrated in Figure 1.2. In comparison with a decomposable legacy IS, only user interfaces, UI_i, and system interfaces, SI_i, are separate modules. The applications and database service are not separable since their structure is more complex, not adequately engineered in accordance with current standards, or is poorly understood. The lack of desirable structure makes analysis and migration more complex and error prone.

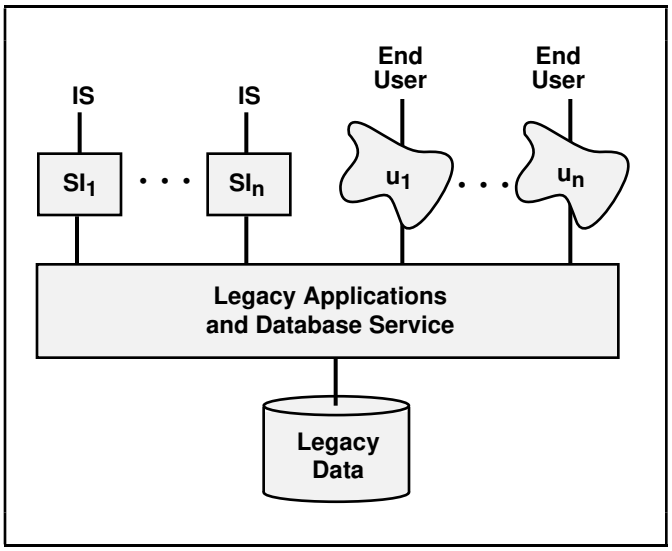


Figure 1.2: Semi-Decomposable Legacy IS Architecture

The worst architecture for migration is a **non-decomposable legacy IS**, illustrated in Figure 1.3. Such ISs are, from a system point of view, black boxes since no functional components are separable. End Users and ISs interact directly with one, apparently unstructured, legacy IS.

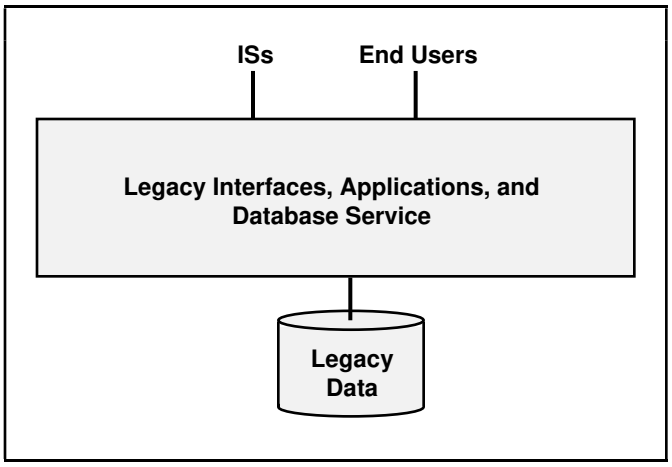


Figure 1.3: Non-Decomposable Legacy IS Architecture

In general, the architecture of most legacy ISs may be neither strictly decomposable, semi-decomposable, nor non-decomposable. During its decade long evolution, a legacy IS may

have had parts added that fall into each architectural category resulting in a hybrid architecture, as illustrated in Figure 1.4. The figure is intended to suggest that some interface and application modules are inseparable from the legacy database service while others are modular and independent.

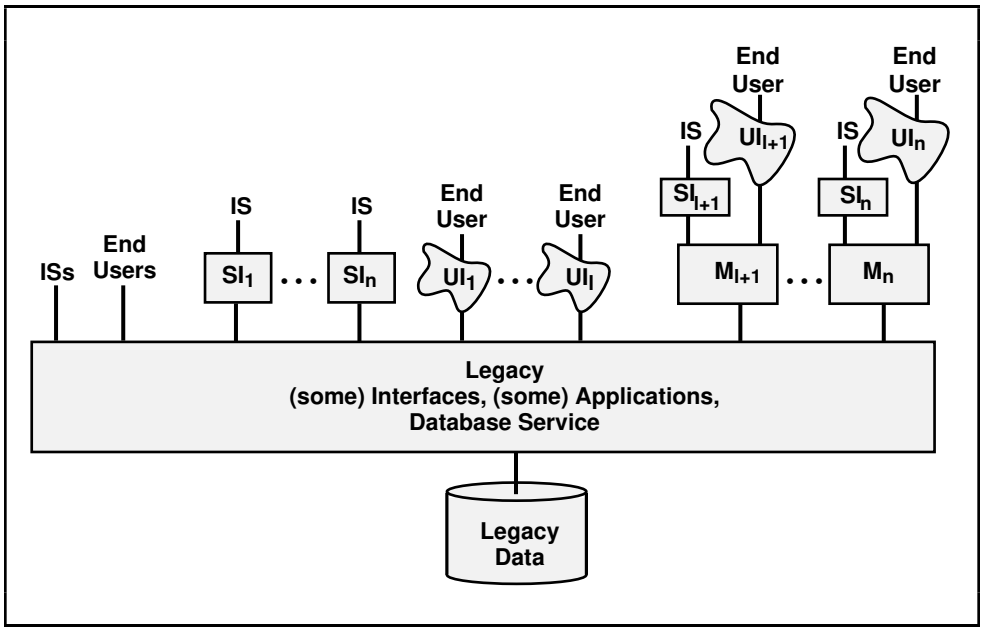


Figure 1.4: Hybrid Legacy IS Architecture

A Chicken Little legacy IS migration involves iteratively selecting and migrating parts of the legacy IS to become new parts of the iteratively constructed target IS. During the migration, the legacy IS and the target IS form a *composite IS* which collectively provides the mission critical IS function. In the composite IS, the legacy IS and target IS are connected by a gateway, as illustrated in Figure 1.5.

Gateways play the key role in the migration architectures described in this paper. By **gateway** we mean a software module introduced between operational software components to mediate between them. Given its controlling position, it can mediate many things (i.e., play many roles). One role is to insulate some components from changes being made to others. In Figure 1.5, the gateway makes any change to the legacy IS transparent to the legacy user interface (UI). That is, the gateway maintains the interface that the UI expects of the legacy IS even though the legacy IS is being changed "behind the scenes." This transparency permits us to alter one part of the legacy IS at a time. This capability is critical to the Chicken Little strategy. As the target graphical user interface (GUI), in Figure 1.5, is iteratively introduced, the gateway makes transparent to the GUI and UI whether the legacy IS or target IS or both are supporting a particular function. Hence, the gateway can insulate a component that is not being changed (e.g., the UI) from changes that are being made (e.g., migrating the legacy database to the target database).

A second gateway function is to translate the requests and data between the mediated components (e.g., UI calls to target IS calls, target IS data to legacy UI formats). Gateways are widely used for the two purposes stated above. Few if any gateways provide the third, critical function of coordination between the mediated components. For example, coordination for update consistency may be required for an update request that the gateway

directs to the legacy IS and to the target IS. In the migration methods presented, we specifically define this critical coordination function of gateways.

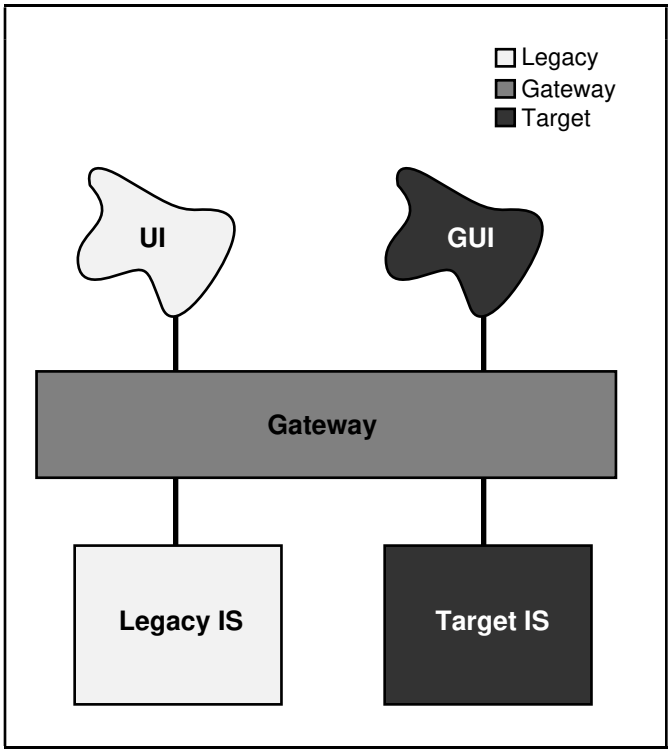


Figure 1.5: IS Migration Architecture

The placement of the gateway is a critical factor that affects the complexity (or simplicity) of the migration architecture, the gateway, and the migration method. In the best case, the decomposable legacy IS, the gateway can be placed between the application modules (M_j) and the legacy database service, illustrated on the right hand side of Figure 1.6 (for simplicity, target ISs are not shown). In this case, we call it a **database gateway** since it encapsulates the entire database service and database from the perspective of the application modules. For the semi-decomposable legacy IS, the lowest the gateway can be placed is between the interfaces and the rest of the legacy IS (i.e., applications, database service, and database), illustrated in the center of Figure 1.6 (and in Figure 1.5). It is called an **application gateway** since it encapsulates from the applications down, from the perspective of the interfaces. Due to the functionality it encapsulates, an application gateway can be considerably more complex than a database gateway. In general, the higher up in the architecture the gateway is placed, the more functionality it encapsulates and the greater the complexity of the gateway. Finally, an **IS gateway** encapsulates the entire legacy IS in the case of the non-decomposable legacy IS. Hence, it is the most complex.

1.3 Methods

A *migration method* consists of a number of migration steps that together achieve the desired migration. A step is responsible for specific aspects of the migration (e.g., database, application, interface). Each legacy IS and its operational context, both business and technical, pose unique and frequently mutually inconsistent, migration requirements that, in turn, require a unique migration method. The migration methods presented in this

paper are tailorable to address these requirements, which include the following:

- Migrate in place.
- Ensure continuous, safe, reliable, robust, ready access to mission critical functions and information at performance levels adequate to support the business's workload.
- Make as many fixes, improvements, and enhancements as is reasonable, to address current and anticipated requirements.
- Make as few change as possible to reduce migration complexity and risk.
- Alter the legacy code as little as possible to minimize risk.
- Establish as much flexibility as possible to facilitate future evolution.
- Minimize the potential negative impacts of change, including those on users, applications, databases, and, particulalrly, on the on-going operation of the mission critical IS.
- Maximize the benefits of modern technology and methods.

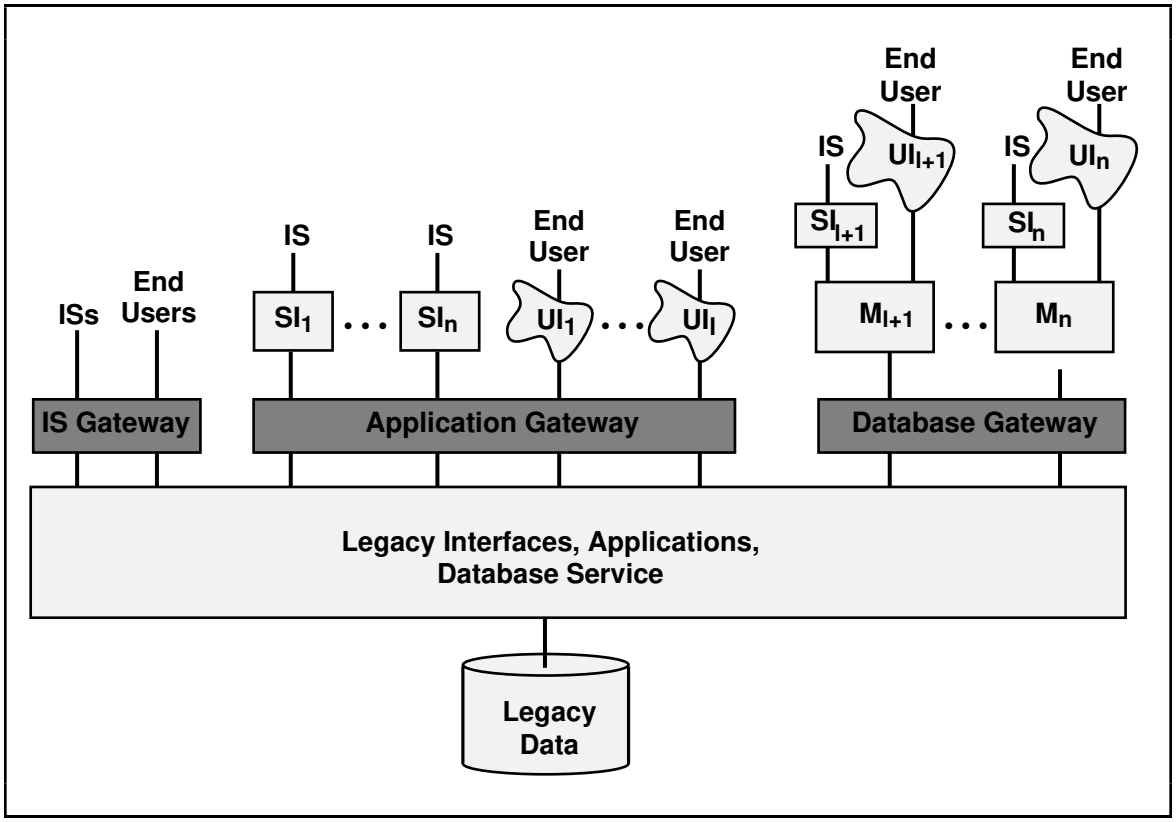


Figure 1.6: Gateway Types and Placements

The iterative nature of the Chicken Little strategy leads to two ways to reduce risk. First, there must always be a fail safe fall back position, should any incremental step fail.

Second, the increment size must be chosen to make the risk of the current step effectively zero. We offer no new ways of evaluating such risks but we do emphasize the importance of risk evaluation and avoidance.

The greater the independence of the steps, the greater the flexibility for adapting a migration method to specific migration requirements, changing requirements, and mistakes. Independent steps can proceed independently (e.g., in any order). Gateways are one of the primary means of providing independence between steps, since they can encapsulate system components that are undergoing change behind an unchanging interface.

This paper presents a set of migration methods that apply to all legacy ISs. Each method consists of five basic steps:

- Iteratively migrate the computing environment.
- Iteratively migrate the legacy applications.
- Iteratively migrate the legacy data.
- Iteratively migrate the user and system interfaces.
- Iteratively cut-over from the legacy to the target components.

Migrating the computing environment, the applications, and the database are obvious steps. The final two steps are less obvious and require some explanation.

Interface Migration and Gateways

First, let's consider interface migration. Legacy ISs are, by definition, mission critical. They contain key corporate resources. The user and system interfaces control all uses of the system and all access to those resources. Hence, IS interfaces are as critical as the databases and the applications. Errors that originate in the interfaces can significantly negatively affect the viability of the content and performance of an IS and of all the ISs and people that interact with it. Generally, user interfaces significantly affect the working environment and productivity of a large number of people (e.g., bank or telephone service order clerks). The system interfaces significantly influence the efficiency of all current and future ISs that interact with the target IS. Interfaces are critical before, during, and after the legacy IS migration. Indeed, the success of the migration depends critically on the interfaces to the composite IS that exists during the migration. Hence, interface migration should be considered equally with database and applications migration. This makes sense technically, since interfaces, databases, and applications have distinct technical challenges and supporting technologies.[4] Interface migration can be used to implement corporate-wide interface improvements (e.g., TTY to GUI) and standardization that is currently being pursued in most IS organizations. Interface migration can provide a basis for, and many of the benefits of, IS integration and interoperability before their being provided at the database and applications level.

---

[4] The separation of interface, database, and application issues and technologies is part of a trend towards next generation information systems technologies, architectures, and methods. The trend is to separate as many aspects as possible to achieve, amongst other goals, greater modularity, flexibility, reusability, portability. This is widely discussed in terms of Enterprise Information Architectures and middleware [BROD92a].

During migration, legacy IS modules and their interfaces can be operational simultaneously with the corresponding target IS modules and their interfaces. This may require that the desktop machine (e.g., PC) of the target environment emulate a *dumb terminal* interface. For IBM legacy code, the PC must include 3270 emulation, a widely available feature. As a result, The user interface could get ugly during the migration with GUI windows existing simultaneously with 3270 windows on the PC. These problems can be addressed by an interface gateway (i.e., a single user interface that can help to simplify the multiple interfaces of the composite IS). It can also hide the fact that the functions are being supported by the legacy IS, the new IS, or both. This helps to insulate IS users from changes in the user interface as well as IS changes. The interface gateway permits the changes to be phased in at a rate appropriate to the user community. Interface gateways can also be used to introduce, temporarily, functionality and extensions intended for the target IS but added earlier to gain some benefits (e.g., essential edit checks not in the legacy interfaces and not yet available in the target interfaces). This adds alternatives hence flexibility to the migration methods.

A key to a successful interface migration is the *interface gateway.* It insulates all end users and interfacing systems from any negative effects of the migration, as illustrated in Figure 1.7. An interface gateway captures user and system interface calls to some applications and then translates and re-directs them to other applications. It also accepts the corresponding application responses and translates, integrates and re-directs them to the calling interface. For some legacy interfaces, this means capturing TTY keystrokes and mapping them into GUI interfaces or directly to applications. An interface gateway provides more independence between the interfaces and the applications thus adding to the flexibility of the corresponding migration methods. Besides gateway functionality, it could provide a complete user interface development and management environment. It might also support migration with versioning and other functionality. Finally, as suggested in Section 9, the interface gateway can be maintained as part of the target IS architecture to support future interface migration and evolution.

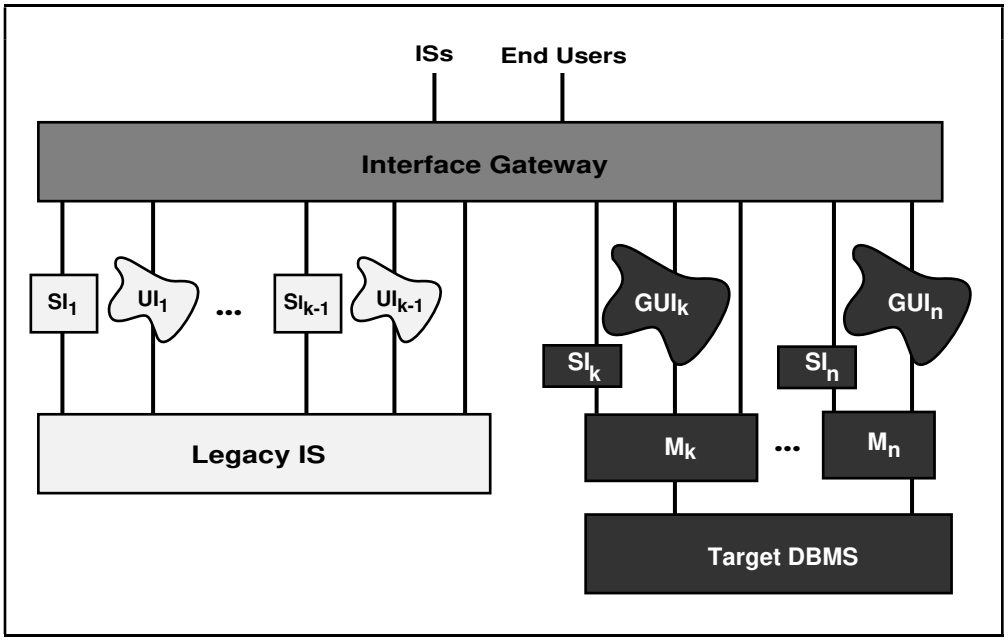


Figure 1.7: Interface Gateway

Cut-Over

We end this subsection by considering cut-over. We use the term **cut-over** to refer to the process of switching from the legacy IS to the target IS. For example, whereas the application migration step is used to design, build, and install target applications, the cut-over step is used to iteratively cut-over operations from the legacy applications to the target applications, application by application, site by site, and user by user according to specific requirements.

Although there could be a cut-over phase for each of the above four steps, the size and complexity of the ISs we are considering can warrant a separate cut-over step. In large organizations, such as banks and telecommunications companies, IS cut-over may involve hundreds of sites, hundreds of users, and hundreds of versions of legacy and target database, application, and interface modules. Target modules may be ready months or years before the target environment is in place or before all end users are prepared for the change. The cut-over step involves coordinating all these components so that the composite IS continues to meet the mission critical IS requirements. This complexity alone can warrant a separate cut-over step. The size of the cut-over step may require that it be iterative (i.e., done in appropriately sized chunks), consistent with the Chicken Little strategy. It can also be optimal to proceed with all steps in parallel. For example, some hardware and some software in some sites can be cut-over for some users while the corresponding legacy components are still in operation. Iterative and parallel cut-over increases the flexibility of the methods but also increases the complexity of the cut-over procedure, further motivating cut-over as a separate step.

A significant challenge for most migration methods is to plan, manage, and modify, as needed, the steps and their interactions. A related challenge is to achieve a migration plan that is adequately coordinated to iterate and parallelize the migration steps. The methods presented in this paper are intended to provide this flexibility so that they can be tailored to the requirements and available resources.

In Section 2, we develop and present the simplest method that applies to decomposable legacy ISs. We illustrate this method in Section 3, with a case study for a legacy banking system from a major bank. For legacy ISs with more complex requirements and architectures (e.g., semi-decomposable, non-decomposable, general case), we extend the simple method in subsequent sections. We also add details that could have been included earlier but were deferred to simplify the initial descriptions. In Section 4, we extend the method to semi-decomposable legacy ISs and illustrate it, in Section 5, by a case study for a legacy telecommunications facilities management system from a large telephone company. In Section 6, we further extend the method to non-decomposable legacy ISs. Finally, in Section 7, we present the migration method for the general case.

The legacy ISs presented in the case studies are typical of hundreds of such ISs known to the authors. They provide excellent examples of legacy IS migration problems and challenges. Although legacy IS re-engineering and migration are frequently discussed, there are few, if any, effective migration methods, tools, or techniques. In this regard, we list, in Section 8, desirable tools and needed research that would facilitate the application of Chicken Little to real legacy ISs. We conclude the paper with summary comments and an epilogue on the status of the case studies.

2. MIGRATING DECOMPOSABLE LEGACY ISs

In this section, we present a Chicken Little method for migrating legacy ISs that are decomposable, or that can be restructured accordingly. Our goal is a target IS that is decomposable and is in our target environment (e.g., *rightsized* computers, a client-server architecture, modern software). Decomposability in the target IS is intended to facilitate future change to avoid future legacy IS problems.

The method for migrating decomposable legacy ISs involves a forward migration method, described in Section 2.1, and a backward migration method, described in Section 2.2. On their own, the forward and backward migration methods are not widely applicable. However, they are fundamental to the migration methods presented in the rest of the paper.

2.1 Forward Migration Method For Decomposable Legacy ISs

This sub-section presents the method for migrating decomposable legacy ISs for which the database can be migrated in one initial, Cold Turkey step. It involves a **forward database gateway**. This facilitates a Chicken Little migration of the applications and their interfaces *after* the Cold Turkey database migration. It is called a forward migration since it migrates unchanged legacy applications *forward* onto a modern DBMS and then migrates the applications. As discussed below, a Cold Turkey database migration and other limitations may render the forward migration method inappropriate for most migrations.

Step F(orward)1 Iteratively install the target environment.

Identify the requirements for the target environment based on the total target IS requirements. Select and test the environment. Install the environment including a desktop computer for each target IS user (e.g., bank clerk, telephone service order clerk) and appropriate server machines. This requires replacing a dumb terminal with a PC or workstation and connecting them with a local area network. Such a move to desktop, client-server computing is currently being studied in most IS shops and is being implemented in many. This facilitates the construction of GUI programs, necessary in subsequent steps, and off-loading code from a server machine, where MIPS are typically expensive, to a client machine, where MIPS are essentially free.

Step F1 involves significant changes in (i) hardware and software, (ii) applications, development, and maintenance architectures, and (iii) users and management. These changes may require significant investments and time. Hence, Step F1 may prolong the entire migration. Following the premise underlying Chicken Little of migrating in small, iterative steps, Step F1 can begin at any time and can be taken in steps appropriate to the context. That is, install a few PCs at a time and deploy new software at a rate appropriate to the context.

Step F2 Analyze the legacy IS.

Understand the legacy IS in detail. This corresponds to writing requirements for the legacy IS and for the target IS. Sources for the understanding include documentation, if any exists, the code[5], and knowledge of the system and its use from people who support, manage, and use the legacy IS. The tendency to include new target IS requirements must be

---

[5] Existing tools can assist with this step (e.g., Reasoning Systems' Refine and Bachman Information Systems' The Analyst for code analysis.

managed due to the consequent increase in complexity and risk of failure.

Step F3 Decompose the legacy IS.

Modify the legacy IS to ensure that it is decomposable. Dependencies between modules, such as procedure calls, must be removed. There must be well-defined interfaces between the modules and the database services. The cost of this step depends on the current structure of the legacy IS. If such a restructuring is not possible, other migration methods may apply.

Step F4 Design the target applications and interfaces.

Design and specify the target IS. This requires designing the software architecture of the target IS, illustrated in Figure 2.1. It includes a modern DBMS and target database that can be centralized, on one server machine, or distributed over the multiple servers in a client-server architecture. It also includes application modules (M_i) each with its corresponding user interface (GUI_i) and, possibly, system interface (SI_i). Design target GUIs and SIs and an interface migration strategy including the decision whether to build an interface gateway. The target modules and interfaces will run on the client machines in the target environment. Following the principle that significant functionality not be added during the migration, legacy and target application module functions are intentionally similar.

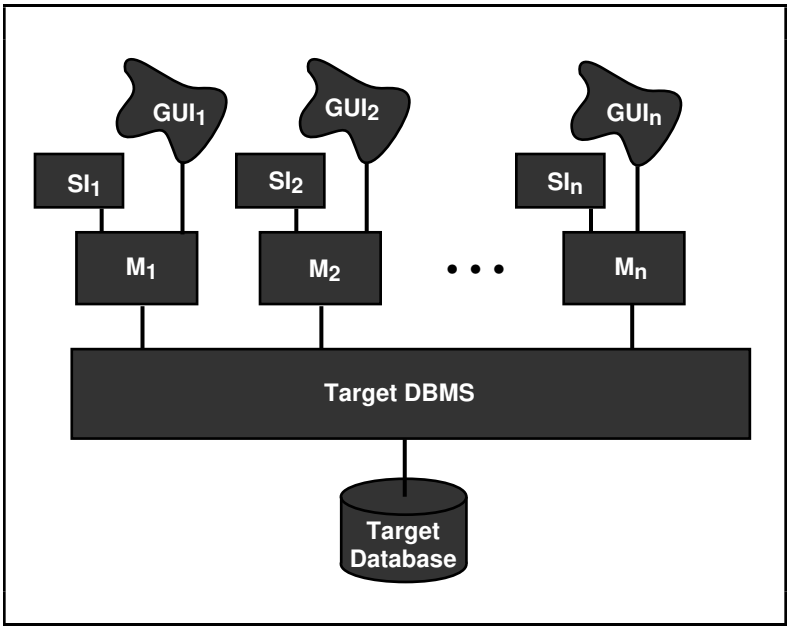


Figure 2.1: Target IS Architecture

Step F5 Design the target database.

Design a relational[6] schema to meet the data requirements of the target IS. This requires an understanding of the target and legacy ISs and uses the results of the previous steps. Depending on the legacy code and the application requirements, the target database design

6 Unless there are very unusual requirements, we recommend the use of an SQL-based, relational DBMS.

step can be very complex. Legacy IS development techniques can make data definitions and structures difficult to find and understand. Before the *age of databases,* the distinction between data and application code was often blurred. It was also common to distribute data definitions throughout the application[7]. The complexity of this step can be large enough to warrant iterating in small increments, as discussed in Section 2.3. Re-engineering tools[8] can be used to extract data definitions from legacy code, to design schema fragments for each increment, and to integrate the schema fragment into a single schema.

Due to the critical role of data in any IS, this step provides benefits whether or not the legacy IS migration proceeds (e.g., a deeper understanding of the IS).

Step F6 Create and install the forward database gateway.

Develop the forward database gateway. The gateway is intended to encapsulate the target DBMS and target database from the legacy applications and to permit the application and database migration steps to proceed independently. The forward gateway is designed so that the legacy applications need not be altered in any way. It includes a translator that captures and converts all legacy database service calls from legacy applications on the mainframe into calls against the modern DBMS on the server machine(s). The conversion may require a complex mapping of the calls (e.g., one to many, many to many, special purpose procedures), and data translation. The gateway must also capture responses from the DBMS, possibly convert them, and direct the result to the appropriate module(s). The gateway can also be used to enhance or correct legacy applications immediately rather than waiting for the application migration step. For example, the data and call translator can introduce new data formats, data edits, and integrity and error checking and correction that will later be done in target applications.

The forward database gateway evolves as the IS migration proceeds. As target applications are cut-over and legacy applications are retired, the gateway's translation and redirection functions are reduced accordingly. To support the mapping and redirection functions, it may be useful to implement mapping tables. Mapping tables are tables or directories that provide a mapping between legacy database service calls and their modern DBMS counterpart(s) as well as between legacy data items and their corresponding target data counterpart(s). The tables can identify when complex mappings (e.g., mapping programs) are required.

Constructing a forward gateway can be very costly. It involves writing the gateway from scratch or tailoring a commercial gateway product[9] to meet the migration requirements. For certain DBMSs, a general purpose forward database gateway can be built. In some cases, constructing a forward database gateway can be very complex due to the low-level legacy database service calls which may have semantics that are unimplementable in SQL. See [DATE87] for a discussion of this point in the context of IMS to SQL conversion. Such cases require a special purpose gateway that handles calls on a case-by-case basis.

---

7 There is currently a trend back to these ideas. Object-oriented and distributed computing principles suggest moving from a single, centralized database schema to distributed class definitions that encapsulate data and functions.

8 For example, Bachman Information Systems' Re-Engineering Product Set.

9 Some DBMS vendors provide forward database gateway functions, other vendors specialize in them. For example, Computer Associates' Transparency Software is intended to provide translation of native or legacy DBMS calls to IMS, VSAM, and DB2 to calls to their CA-Datacom DBMS.

The gateway must be installed in the legacy IS architecture between the application modules and the legacy database service in preparation for the database migration (Step F7). A gateway can significantly impact IS performance. It must be carefully designed and thoroughly tested keeping in mind that the composite IS (the legacy IS plus the target IS) is mission critical and can never fail.

If a forward database gateway is impractical, alternative methods, discussed in subsequent sections, may be applicable.

Step F7 Migrate the legacy database.

Force-fit the legacy application modules to the target database to achieve the architecture illustrated in Figure 2.2. This involves installing the target DBMS on the server machine(s), implementing the schema resulting from Step F5, migrating the legacy database to the target DBMS under the new schema, and using the gateway to support the legacy application calls. Database migration involves downloading, converting, and uploading, possibly large amounts of data. Products[10] support some of these functions. The forward database gateway may be useful in the database migration as it contains relevant mapping and translation information.

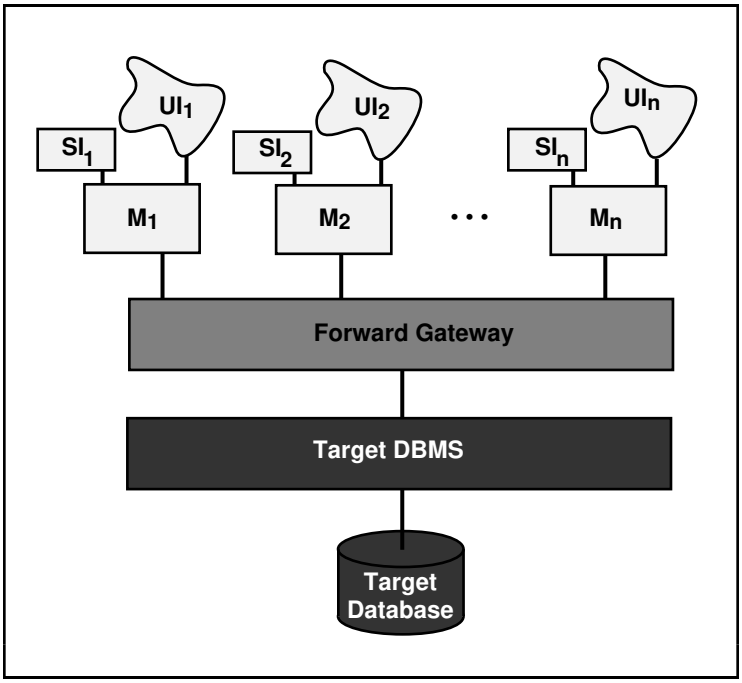


Figure 2.2: Forward Migration Architecture, Initial State

Cold Turkey database migrations may be impractical for several reasons. The legacy database may be so large or complex that there is no effective one-step, Cold Turkey migration method. Even if a method exists, some databases are so large that the time required for migration is greater than that allowable for the system to be non-operational. Such cases may require an iterative database migration method, as described in subsequent

10 For example, IBM's Data Extractor Tool (DXT) and BMC Software's LoadPlus.

sections.

Step F8 Iteratively migrate the legacy applications.

Select and migrate legacy modules, one (or more) at a time. Selection should be based on technical (e.g., simplicity) and organizational (e.g., cost, importance) criteria. This involves rewriting the legacy module(s)[11] (M_i) to run directly against the modern DBMS, as illustrated in Figure 2.3 for modules M_1 through M_j.

The target applications will run on a client machine in the target environment. They can be used to replace the legacy application modules that run on dumb terminals. The remaining legacy modules (M_k through M_n in Figure 2.3) must continue to be used.

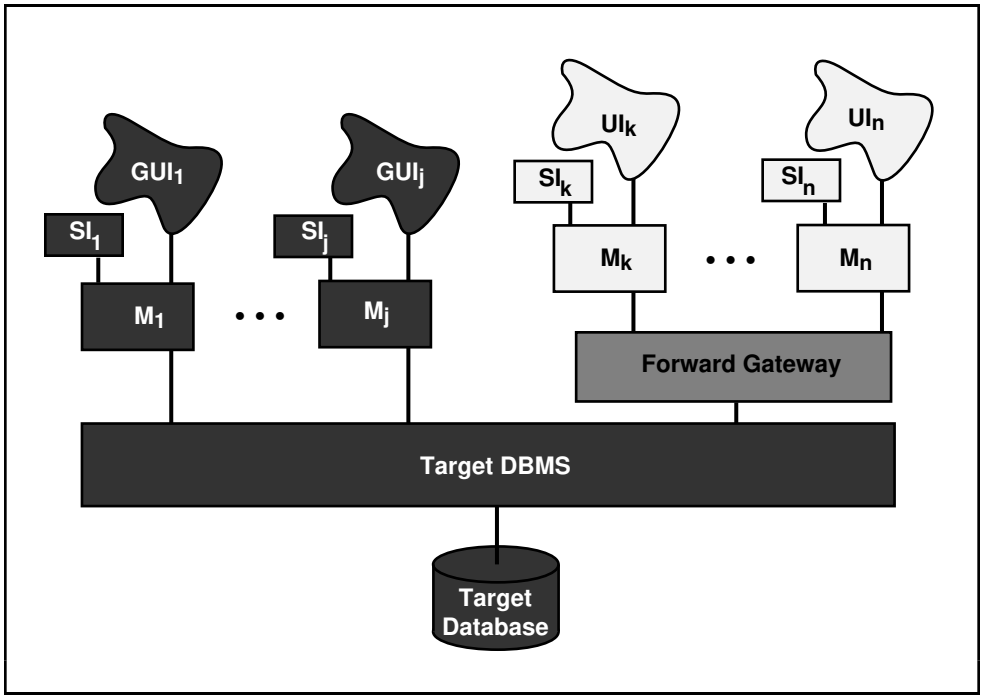


Figure 2.3: Forward Application Migration Architecture, Intermediate State

Step F9 Iteratively migrate the legacy interfaces.

Select and migrate legacy interfaces, one (or more) at a time. Selection should be based on principles similar to those in Step F8. If a 4GL is used which supports application and interface development, interface and application migration might be coordinated. This involves rewriting the legacy user (UI_i) and system interfaces (SI_i) to run directly against the modern DBMS. The target interfaces will run on a client machine in a 4GL/GUI environment on the desktop. They can be used to replace the legacy interfaces that run on dumb terminals. The remaining legacy interfaces (U_k and SI_k through U_n and SI_n in Figure 2.3) must continue to be used. An interface gateway could be used here to support

[11] There is a growing number of products to assist with rewriting legacy code into target applications. For example, Seer Technologies Inc.'s High Productivity System assists in converting IMS applications to SQL.

interface migration.

Step F10 Iteratively cut-over the target IS.

Cutting-over the new IS involves cutting-over operations to the forward gateway, and the migrated target database, applications, and interfaces on client-server machines, and then discarding the legacy components. When the last module is cut-over and no legacy modules are in use, the forward database gateway can be discarded. However, since change is constant and the migration process may take a long time, the gateway may serve future migration and evolution requirements, as discussed in Section 9. Finally, the costly legacy environment, including the mainframe and the dumb terminals, can be discarded. The resulting target IS is illustrated in Figure 2.1.

The cut-over can begin as soon as the database is migrated. It continues as applications are migrated and can be extended indefinitely. For example, it may be economical to run a bank branch or telephone office on the legacy IS for a year after others have migrated if they are to be shut down thus losing the migration investment. Using the forward database gateway, legacy IS modules and their interfaces can be operational simultaneously with the corresponding target IS modules and their interfaces as long as required.

The flexibility, hence applicability, of a migration method increases as the steps become iterative and parallel. In the forward migration method, only four steps (i.e., F1, F8, F9, and F10) were described as iterative, partly to simplify the explanation. More steps could and should be iterative. For example, the database migration step could be made iterative. However, this will complicate the cut-over as well as the gateway that would have to mediate between the diminishing legacy database and the growing target database. In subsequent migration methods in this paper, all steps become iterative.

2.2 Reverse Migration Method For Decomposable Legacy ISs

This sub-section presents the method for migrating decomposable legacy ISs for which the database can be migrated in one final, Cold Turkey step. It involves a **reverse database gateway** that facilitates a Chicken Little migration of the applications and their interfaces *before* the Cold Turkey database migration. It is called a *reverse* migration since it migrates target applications in the reverse direction, back onto the legacy database until it is subsequently migrated. This method permits more time to deal with the database migration. As with the forward migration, a Cold Turkey database migration and other limitations render the reverse migration method of limited applicability. Reverse migration principles are important as they are included in all subsequent migration methods.

In the reverse migration method is similar in most steps to the forward method. We focus here on only those steps that differ.

Step R(everse)1 Iteratively install the target environment. (See Step F1)

Step R2 Analyze the legacy IS. (See Step F2)

Step R3 Decompose the legacy IS. (See Step F3)

Step R4 Design the target applications and interfaces. (See Step F4)

Step R5 Design the target database. (See Step F5)

Step R6 Create and install the reverse database gateway.

Develop the reverse database gateway. The reverse database gateway includes a translator that captures and converts all calls to the modern DBMS from target applications and maps them into calls to the legacy database service. It must also capture, translate, and direct responses from the legacy database service to the appropriate modules.

The functions of and challenges in developing a reverse gateway are similar to those of the forward gateway (e.g., potentially complex mappings). As the IS migration proceeds, the reverse gateway must be evolved. Initially, it may support one target application module. Iteratively, it supports more target application modules until all are supported, thereby completely encapsulating the legacy database service. It contracts as the target applications are migrated to access the target database directly.

As with the forward database gateway, the reverse database gateway may require mapping tables or directories, data and call conversion, and can have its functionality extended. Also, there are many alternatives for constructing a reverse database gateway (e.g., hand-code functions as needed, build a general purpose reverse database gateway, adapt commercial products). Hand coded reverse database gateways may be appropriate for small IS migrations. However, a general purpose reverse database gateway may be more appropriate for large IS migrations. Some products provide reverse database gateway functions[12]. These products do not support data integrity, transactions (i.e., updates), or query optimization. They also require that the calling applications be aware that they are calling a gateway and not a DBMS directly. This will require changes to all target applications once the gateway is removed.

The reverse database gateway must be installed in the migration architecture between the target application modules and the legacy database service, as illustrated in Figure 2.4. It then becomes a vital component in the migration and operation of the mission critical IS.

Step R7 Iteratively migrate the legacy applications. (See Step F8)

Using the reverse gateway, map the target application modules onto the legacy database to achieve the architecture illustrated in Figure 2.4. Select and migrate legacy modules one (or more) at a time. This step is similar to Step F8 in the forward migration method except that, initially, target application modules run against the reverse gateway.

The target applications will run on a client machine in the target environment on the desktop. They can be used to replace the legacy application modules. The remaining legacy modules (M_k through M_n in Figure 2.4) must continue to be used until they too are migrated.

Step R8 Iteratively migrate the legacy interfaces. (See Step F9)

---

12 Some DBMS vendors provide reverse gateway functions, other vendors specialize in them. For example, SQL Solutions' RMS Gateway product provides translation between DEC's RMS (files) and various RDBSMs. Oracle's SQL*Net and SQL*Connect provide translation from SQL to DB2, SQL/DS, RMS (files), and IMS. Information Builders Inc.'s Enterprise Data Access (EDA) provides translation between more than 50 DBMSs and file systems in client-server environments. Cross Access Corp.'s Cross Access product provides similar translation between more than 15 DBMSs and file systems.Apertus Technologies' Enterpise/Access product will povide reverse gateway services.

Step R9 Migrate the legacy database. (See Step F7)

Step R10 Iteratively cut-over the target IS. (See Step F10)

This step is similar to the forward migration Step F10 except that the reverse migration cut-over is more constrained and, consequently, of higher risk. The forward migration architecture can simultaneously support legacy and target applications. This provides flexibility as to how and when to cut-over the IS. In the reverse migration method, once the legacy database is migrated, the legacy applications can no longer be used. Hence, the database, applications, and interface cut-overs must be coordinated and must conclude simultaneously. This may place severe constraints on the cut-over since the IS is mission critical and can be non-operational for only a very short time.

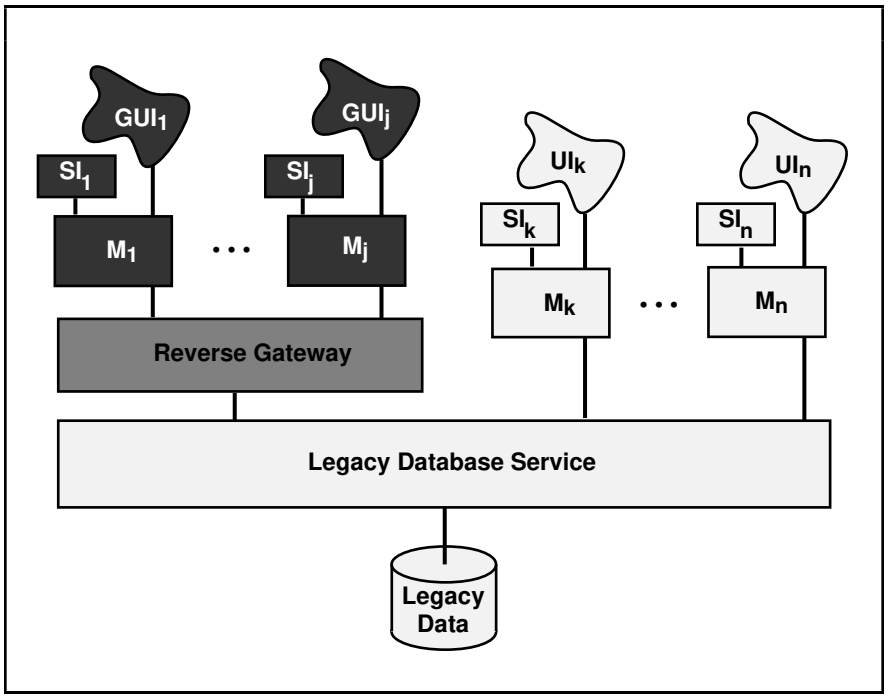


Figure 2.4: Reverse Migration Architecture, Intermediate State

2.3 General Migration Method For Decomposable Legacy ISs

This sub-section presents a migration method for all decomposable legacy ISs. It uses a reverse database gateway and a forward database gateway, which permits all migration steps to be iterative and parallel. Following our Chicken Little strategy, IS migration can proceed by selecting, as iteration increments, any appropriate subset of IS functions and corresponding data (or *vice versa*).

For simplicity, we describe only the key differences with the forward and reverse migration methods already described. These include, the gateway (Step D6), and the database, application, and interface migration steps (Steps D7, D8, and D9). Making Steps D2 through D5 iterative is a challenge left to the reader, as is their coordination in the cut-over, Step D10.

Step D(ecomposable)1 Iteratively install the target environment. (See Step R1)

Step D2 Iteratively analyze the legacy IS. (See Step R2)

Step D3 Iteratively decompose the legacy IS. (See Step R3)

Step D4 Iteratively design the target applications and interfaces. (See Step R4)

Step D5 Iteratively design the target database. (See Step R5)

Logically partition the legacy database to facilitate iterative database migration. Identify subsets of the legacy database that are sufficiently separable to be migrated independently to the target database. For each subset, design the target database schema (i.e., apply Step R5) so that the subset can be integrated into the target database.

Step D6 Iteratively create and install the database gateway. (See Steps F6 and R6)

Unlike the reverse and forward methods, this migration may require simultaneously supporting portions of the corresponding legacy and target applications and databases. During the migration, the operational mission critical IS will be some composite of legacy and target ISs, as illustrated in Figure 2.5. The gateway must support this composite. For example, it must ensure that a degree of update integrity be maintained on any part of the legacy database that is replicated in the target database. For example, if a data item in the legacy database is updated, the corresponding target data item may be required to be updated accordingly within a given time. Such correspondences must be identified, expressed in inter-database dependencies (e.g., conditions under which legacy database updates must be reflected in the target database, and *vice versa*), and maintained by the gateway. This coordination role of the gateway is similar to the transaction management role of a DBMS.

This step is iterative in that the gateway functions vary throughout the migration. At the beginning of the migration, when the target database is empty, and at the end, when the legacy database is empty, there is little for the gateway to do. When there are many legacy and target components operating simultaneously, the gateway must maintain all the inter-database dependencies. Failure to do so at all or within adequate performance bounds may cause the mission critical, composite IS to fail.

The gateway consists of at least four components: the forward database gateway, the reverse database gateway, a mapping table, and a coordinator. Since the potential mappings are more complex than in the forward and reverse database gateways, mapping tables will be correspondingly more sophisticated. It may be useful to maintain other descriptive data to assist with the migration. This meta-data may form a small migration database that could be supported by the target DBMS. If the database mappings are sufficiently complex, the migration database may contain a schema that integrates those of the legacy and target databases, as is done in distributed DBMSs [OZSU91]. Indeed, the gateway bears strong similarity to a general purpose distributed DBMS.

The coordinator manages all gateway functions. On the basis of the location of the data being accessed and on the relevant inter-database dependencies, all given in the migration database, the coordinator must map calls from the legacy and target applications to one or more of the legacy database, the target database, the reverse gateway or the forward gateway. The alternatives are illustrated in Figure 2.5. Calls from legacy modules M_k through M_n can be directed to the legacy database service, without translation, and to the target DBMS via the forward database gateway. Calls from target modules M_1 through M_j

can be directed to the legacy database service via the reverse database gateway and to the target DBMS, without translation. The gateway may also need to combine responses from both database services and map them to either or both legacy and target applications. The most challenging requirement for the coordinator is to ensure the inter-database dependencies for updates as well as for queries mixed with the updates. This may require a two-phase commit (2PC) protocol [SKEE82] to be used by both the legacy and target DBMSs and by the gateway coordinator, as is done in distributed DBMSs [ELMA92, OZSU91].

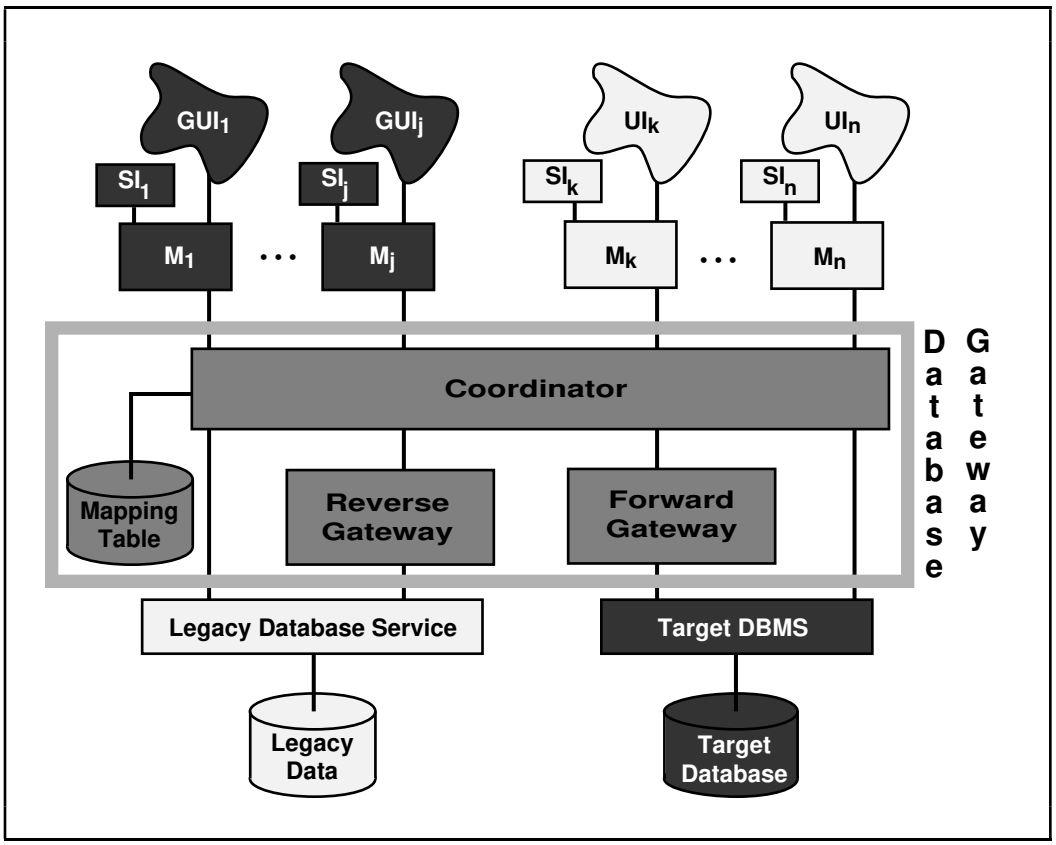


Figure 2.5: Decomposable Legacy IS Migration Architecture

In an IBM environment, CICS can perform the coordinator role and both DB2 and IMS support 2PC; hence distributed transactions are easily supported. However, most legacy database services do not support 2PC. To guarantee that the inter-database dependencies are maintained, 2PC can be hand coded into the application modules. This exotic and difficult work-around is described in [BREI90]. Alternatively, the user can decompose a distributed transaction into two transactions, each updating only one database. If either transaction fails to commit, application logic can perform a compensating transaction to return the database to a consistent state. Compensating transactions are discussed in [GARC87, WACH92]. Another alternative that alleviates the application programmers from such concerns is to develop distributed transaction support in the coordinator, based on existing and special purpose components (e.g., build the coordinator on the target DBMS that might also support distributed transactions). Besides providing 2PC protocols and distributed transaction management, the legacy database service must also be augmented with other transaction support including transaction commit and abort, rollback,

and compensation. This will be one of the most complex technical challenges in the migration and should be left to expert DBMS developers. It is this coordinator function that current gateway products do not support. Although costly and risky, the benefits of building a coordinator may be more considerable than might initially be thought. The longer the gateway is used, the greater the benefits of implementing distributed transactions support, since it is amortized over a longer productive period. Later in this paper, we suggest that IS migration will become a way of life. Hence, distributed, flexible transaction support, as described above, will become a critical component of the target environment. No such products exist.

Step D7 Iteratively migrate the legacy database. (See Step R9)

Select one (or more) independent legacy database subsets (identified in Step D5), based on technical and organizational criteria. Implement the corresponding schema in the target DBMS (e.g., by iteratively augmenting the current target schema). Migrate the corresponding legacy database subset to the target DBMS. This might be aided with several potential migration or downloading tools (e.g., in the legacy database service, in the target DBMS, or special purpose products). The gateway could be extended to support database migration since the migration database could contain the relevant meta-data for translation. The gateway must be enhanced to accommodate any new inter-database dependencies that may have arisen from the migration of the current database subset.

Database migration can use the following simple method. When the migration of some subset of the database is attempted, there are K old modules and N - K new ones . The new applications use a reverse gateway to convert as needed from SQL to the legacy database service. The old applications use the forward gateway when needed to talk to the SQL DBMS. Now, introduce a target distributed DBMS that supports *fragments* for database tables. Hence, each table can be distributed, and a distribution criteria determines where individual records reside. For example, the following distribution criteria places young employees on machine 1 and old employees on machine 2.

```
distribute EMP      to machine-1 where EMP.age < 30

                    to machine-2 where EMP.age >= 30
```

Further suppose the distributed DBMS supports distributed transactions through 2PC. Lastly, most distributed DBMSs allow data to be stored in tables managed by other vendor's single-site DBMSs. This is accomplished by an SQL reverse gateway within the distributed DBMS that translates from SQL to the foreign vendor’s protocol.

Such software makes migration a breeze. Bring up the distributed DBMS with the initial distribution criteria:

- old system: everything
- new system: nothing

Legacy IS transactions are supported by converting legacy database accesses to SQL accesses that are processed by the distributed DBMS. If necessary, the distributed DBMS can route accesses to data not yet migrated through the reverse gateway to the old DBMS.

Over time the migration is accomplished by changing the distribution criteria in small increments until it is finally:

- old system: nothing
- new system: everything

The cut-over Step D10 must be invoked to support the related database cut-over that brings the migrated database into operational use.

Step D8 Iteratively migrate the legacy applications. (See Step R7)

Select and migrate legacy modules one (or more) at a time similar to Step R7. A target module will run against the gateway until the gateway is no longer required (e.g., the corresponding target database has been migrated and no coordination is required). Then, the target module can be migrated from the gateway to the target DBMS.

Step D9 Iteratively migrate the legacy interfaces. (See Step R8)

Step D10 Iteratively cut-over the target IS. (See Step R10)

Coordinate all the iterative, and possibly parallel, migrations of subsets of the legacy IS (e.g., environment, database, applications, and interfaces) making them operational while ensuring that the composite IS meets its mission critical requirements. This step is similar in nature to the corresponding forward and reverse migration steps, Steps F10 and R10. This step offers a wider range of alternatives to avoid problems that arise in those steps.

A fundamental difference with the forward and reverse migrations is the need to coordinate updates between the legacy and target databases. Throughout the migration, some, or all, of the legacy database on the mainframe will be operational simultaneously with the target database on the database server(s). Hence, there may be distributed transactions that perform updates in both systems using a 2PC protocol. At the end of the cut-over process, the distributed DBMS (suggested in Step D7) can be discarded or used in cutting over the next portion of the database.

The cut-over must deal with iteratively retiring subsets of the legacy IS that have been migrated after the corresponding target IS subset is operational. The legacy subsets should be retired only if they are of no further use. This permits the gateway to be simplified accordingly. A legacy database subset is no longer of use only when it is strictly independent from all other database subsets and no legacy application accesses it. Due to the interdependence within legacy databases and between legacy databases and their applications, this may be difficult to judge. For example, what may appear as a two or more logically distinct data groupings may be stored physically as one highly interdependent data and index structure. Such considerations significantly complicate the cut-over step.

3. CASE STUDY 1 MIGRATING CMS

We had the opportunity to construct a migration plan for the Cash Management System (CMS) for a large money center bank. We used the occasion to develop and validate the above methods. This section describes CMS, our analysis, and our migration plan.

3.1 CMS

CMS supports check processing and other specialized services for large corporate customers. One service CMS provides is *zero balance accounts* for which the bank will notify the customer of all the checks that are processed during a given day, and allow the customer to cover the exact amount of these checks with a single deposit. Hence, the customer applies the minimum possible capital to cover his liabilities, and only at the exact time that the capital is needed.

A second CMS service is the *reconciliation* of cleared checks. A customer can provide the bank with an electronic feed of all the checks that he writes each day. The bank will match the issued checks against those that clear, and provide the customer with an electronic feed that indicates all checks that have cleared, as well as those that are still pending.

A third service supported by CMS is electronic funds transfers between customer accounts. When the initiator or recipient of the transfer is another bank, the funds must be electronically received from or transmitted to the other bank. This requires connection to several electronic money transfer systems (e.g., Swift) as well as to the Federal Reserve bank. CMS also supports *lock box* operations in which U.S. mail is received from a post office box and is opened; the checks are deposited for the customer; and an accounting is rendered. Such a service is appropriate for a customer who receives large numbers of checks in the mail, such as a large landlord or a utility company. A final example CMS supported service is on-line inquiry and reporting of account status by customers as well as on-line transactions such as the previously discussed funds transfer.

CMS includes 40 separate software modules that perform these and other functions, totaling approximately $8*10^6$ lines of code. Most of the code runs in a COBOL/CICS/VSAM environment. However, the connection to the Federal Reserve bank is implemented on a Tandem machine using TAL. Lockbox operations are provided on a DEC VAX and are written in C. These additional environments exist since the bank bought external software packages, and then acquired the hardware to run them.

The majority of CMS was written in 1981. It has grown to process between 1 and 2 million checks in a batch processing run each night and approximately 300,000 on-line transactions each day. Most of CMS runs on an IBM 3090/400J with 120 spindles of DASD. Total on-line storage exceeds 100 gigabytes.

CMS is probably too complex for any group to understand in its entirety. Much of the CMS code provides interfaces to ISs elsewhere in the bank or in other organizations. These interfaces are not key CMS functions. To reduce the migration problem to one of manageable proportions, we removed from consideration modules that are not within the core function of CMS.

After discussion with application experts, we concluded that core functions of CMS were supported by the following three subsystems:

- Xcheck -- 1,000,000 lines of COBOL
- Xtransfer -- 200,000 lines of COBOL
- Xcash -- 500,000 lines of COBOL

Xcheck provides batch check processing and reconciliation mentioned above. It also supports on-line transactions submitted by internal bank personnel from synchronous 3270 terminals. Xtransfer supports electronics funds transfer and delivers transactions to Xcheck when account updates are required. Xcash is an on-line system that supports inquiry and update to the Xcheck VSAM files from dial-up asynchronous terminals on customer premises. The terminals are typically IBM PCs.

The decision was made that the software supporting the paper processing operation (check sorting) that occurs before batch processing by Xcheck should be excluded from consideration. Although this is a core function, it is a front-end, easily isolated module that contains vendor-supplied code particular to the check sorting hardware. Because our client had no interest in new check sorters and was happy with this portion of the system, we ignored it.

As a result, we concentrated exclusively on a migration plan for the 1.7M lines of **core** CMS code listed above, ignoring the remaining 6.3M lines of code.

3.2 Analysis of the CMS Core

The next step was a detailed analysis of the CMS core. The purpose was to understand the application structure (i.e., *chunks* of code that are not dependent on the rest of the system). A code analyzer would have helped to extract the desired macro structure. In our exercise, we estimated the required structure by conversations with application experts.

The 1.7M lines of CMS code had the following approximate composition:

Xcash: This system is composed of some 500 modules, each a separate on-line transaction. There is an average of 1000 lines of COBOL to support each transaction. The actual work is performed by Xcheck update routines. An immediate conclusion was that this system should be re-implemented in a 4GL for deployment on a desktop. Compared with COBOL, a modern 4GL should achieve a factor of 20 code reduction. Hence, there should be around 25,000 lines of 4GL.

Some customers will choose to run the 4GL code on a desktop PC. Others will choose not to have bank code running on their machines preferring to use their PCs as dumb terminal emulators. Hence, the bank must supply one or more *application servers* on which the 4GL application will run. Such application servers must be multi-threaded, and a variety of low-cost UNIX servers (*jelly beans*) make ideal candidates for deployment of this sort of code.

Xtransfer: This system provides a data entry facility for an operator to specify the funds transfer transactions, an auditing and message tracking facility, and an interface to several wire service systems. About 15% of this COBOL program is user interface code that should be rewritten in a 4GL; the remainder should be rewritten or ported to the new environment.

Xcheck: This system is the ultimate core of CMS. It consists of two portions, one for batch processing and one for on-line inquiry.

The on-line portion consists of 250 on-line transactions for 3270 terminals on the desks of bank personnel. Each transaction is a separate program, averaging 1000 lines. These programs call a *kernel* collection of programs that access and update the VSAM files. The kernel is a well-structured collection of modules totaling about 32,000 lines of COBOL. This portion of Xcheck should be rewritten in a 4GL. The bank should ensure that a PC is on every employee's desk so that Xcheck need only support a GUI interface for a PC.

The batch processing portion of Xcheck consists of 294 modules, performing the following functions:

- 4 modules access and update VSAM files. These are batch version of the *kernel* mentioned above for the on-line portion. They total approximately 40,000 lines of COBOL.
- 40 modules prepare data for processing. These programs filter data and repair fields in records.
- 130 programs are customer specific report generating and formatting programs.
- 80 programs write internal bank files for audit purposes.
- 40 other programs produce assorted reports.

We were convinced that 250 of these 294 modules perform basic report generation, and should be rewritten using the SQL provided by the target DBMS, plus its report writer (available in all commercial DBMS packages).

As a result, the batch portion of Xcheck consists of about:

- 40,000 lines of kernel code that executes batch updates.
- 40,000 lines of *prep* code that execute before the batch run.
- 250,000 lines of reporting code that should be rewritten.

Lastly, Xcheck appears to have about 300,000 lines of code that serves no current function.

3.3 The CMS Migration Plan

The client was interested in a migration plan for CMS because it was a typical legacy IS. Since CMS costs around $16M per year in hardware and related support costs, management was very interested in *rightsizing*. Moreover, they were eager to move to a modern DBMS and evolve away from the dependence on a single batch run implementing an *old-master / new-master* methodology. Bank customers are eager to know earlier in the day how much money will be coming into their account (from deposits or lockboxes) and how much will be going out (from cleared checks). In effect, customers want an *on-line* transaction system rather than a *batch* transaction system. Such features are incompatible with the existing batch-oriented system.

The client requested that we construct a migration plan satisfying the following constraints, which reflect real financial and business responsibilities in the client environment:

- No migration step could require more than 10 person-years of development.

- No migration step could take more than one calendar year.
- Each migration step had to pay back its cost over a maximum of one year.

We now outline the migration plan that we constructed. Our starting point is the legacy CMS illustrated in Figure 3.1. Following Chicken Little principles, the migration plan decomposed the CMS migration into seven incremental migrations, each for a relatively independent subset of CMS. Each migration applies the migration method for decomposable legacy ISs described in Section 2.3. We focus the description on the seven steps and how they intermix the critical database and application migration steps (Steps T7 and T8).

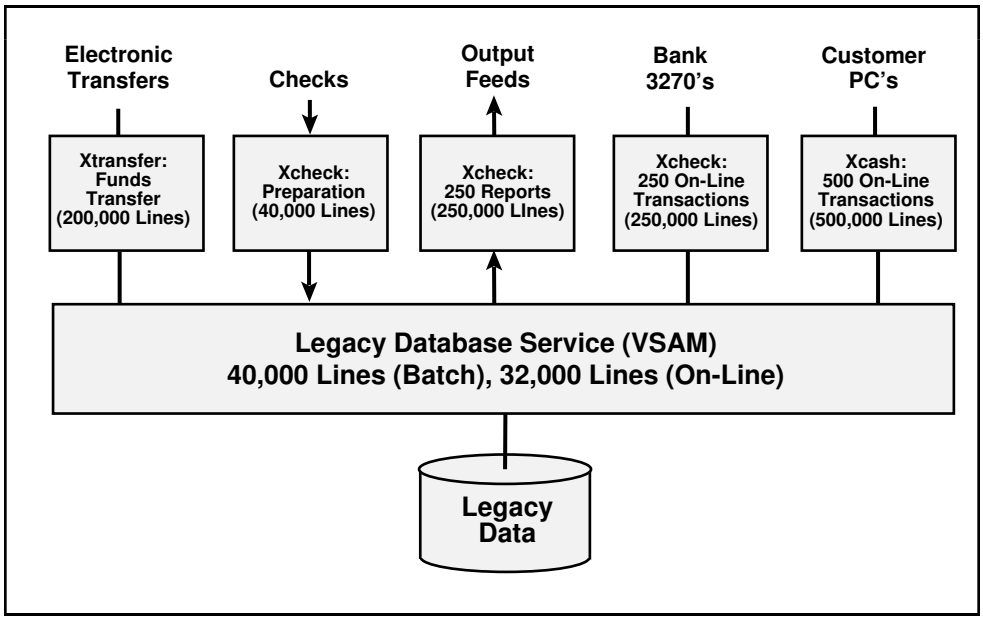


Figure 3.1: Legacy CMS Architecture

Migration 1: Re-specify Xcash in a 4GL.

Xcash should run on client machines in the customer premises. The new 4GL code must be able to submit transactions coded in SQL to Xcheck over an LU 6.2 interface, so that the existing code can perform the actual VSAM inquiries and updates. This requires a forward gateway, as illustrated in Figure 3.2. This migration step requires less than the 10 man-year limit. The bank management agreed to this step based on remaining competitive and since it will substitute cheap *jelly bean* cycles for expensive mainframe cycles.

Migration 2: Rewrite Xtransfer in a 4GL.

This would also be deployed on a PC. This step would be sold to management since it will take less than 5 man years, will lower maintenance costs, and will consolidate the three versions of Xtransfer that are currently in operation.

Migration 3: Rewrite the on-line portion of Xcheck in a 4GL.

Again, the 4GL must be able to connect to the on-line kernel of Xcheck through LU 6.2. This step involves 12,500 lines of 4GL code which should be implementable within the 10

man-year constraint, and would be sold to management based on moving cycles from expensive machines to cheap machines. The result of this migration is illustrated in Figure 3.3.

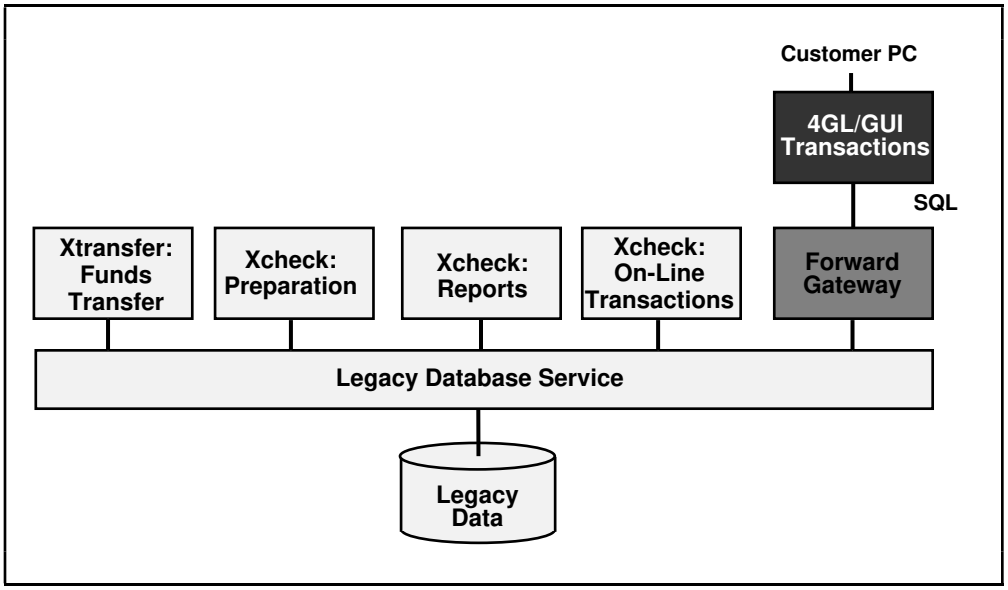


Figure 3.2: CMS After Migration 1

The batch processing code is somewhat more complex to migrate. The on-line kernel and the batch kernel must be combined, moved to a DBMS; and 250 new reports and feeds must be constructed. In addition, the *prep* code must be moved to cheaper hardware. This process was estimated at considerably more than 10 person-years, so it was broken into smaller increments.

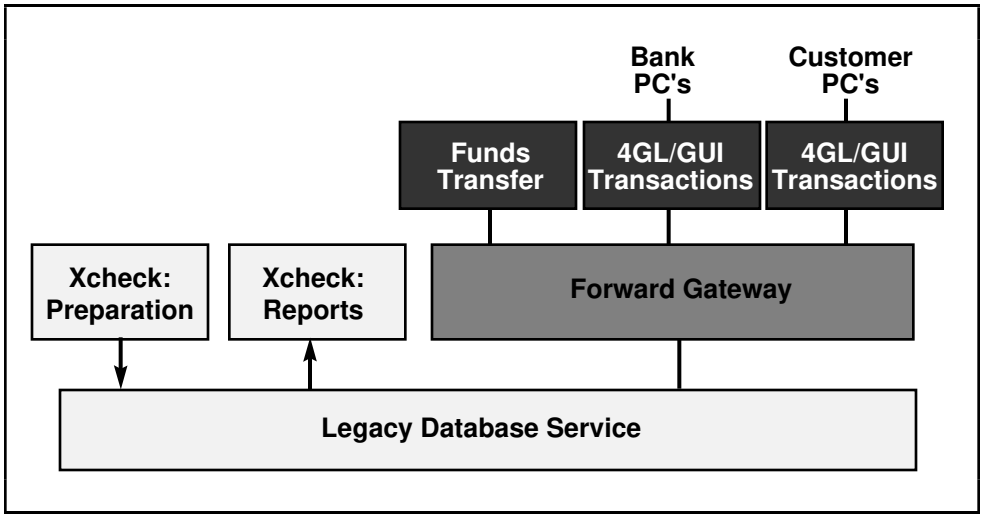


Figure 3.3: CMS After Migration 3

Xcheck performs two services:

- account maintenance
- check reconciliation

After lengthy discussion, it was decided that the migration plan with the least risk entailed migrating these two functions separately. In general, when a large system performs multiple functions, it is a useful strategy to migrate the individual functions separately. Therefore the next migration step was:

Migration 4: Migrate the reconciliation database to an SQL system.

This migration results in a new reconciliation database running in parallel with the legacy database. It requires reverse and forward gateways and a coordinator, as illustrated in Figure 3.4.

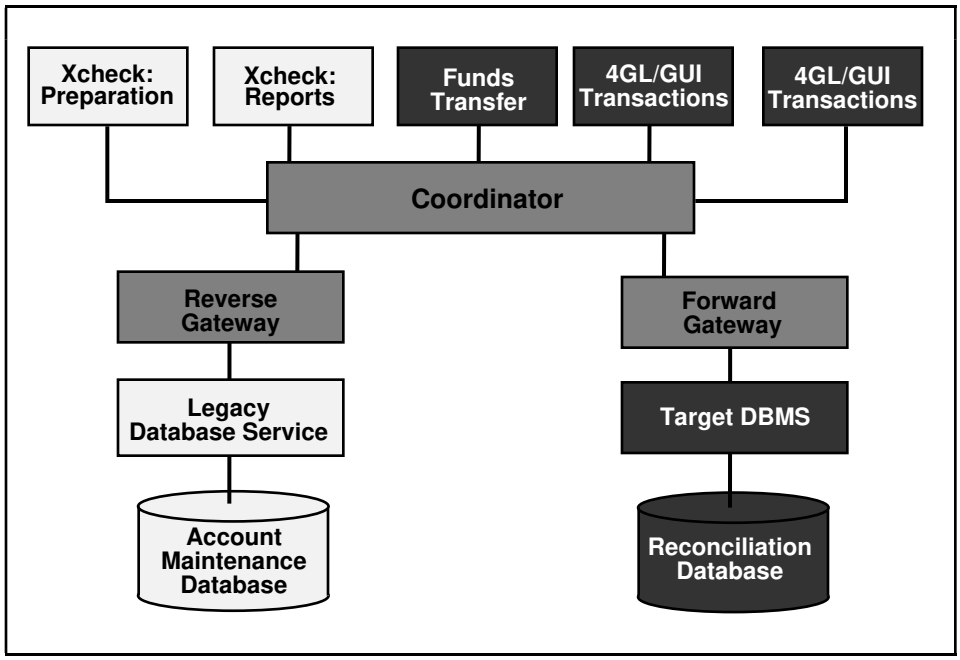


Figure 3.4: CMS After Migration 4

Account balances remain in the legacy VSAM files. The coordinator must ensure that some information in transactions to be diverted to the SQL DBMS must also go into the legacy VSAM files. This can be done since all the reconciliation activity on each account can be put into a single, *aggregate* VSAM transaction that can be inserted into the batch stream. This is an example of the legacy and target databases running in parallel as described in the database migration Step D7. The majority of the database is moved from the old to the new environment. However, the old environment is still the *system of record*. This is ensured by the coordinator.

Migration 5: Rewrite the data feeds and reports using SQL and a report writer.

Data feeds and reports that involve both reconciliation and account balances must be modified to work correctly in the hybrid environment illustrated in Figure 3.4.

We felt that the fourth and fifth migrations could be accomplished in a 10 man-year period. Moreover, the project could be sold to management based on earlier availability of reconciliation information, moving cycles from expensive hardware to cheaper hardware, and moving a portion of the database from expensive mainframe DASD to cheaper DASD.

Migration 6: Migrate the *prep* code to the *jelly bean* environment.

Migrate the 40,000 lines of *prep* code from the mainframe to the cheaper *jelly bean* environment. It is conceivable that the current code could be ported. A more plausible scenario is that it be rewritten as an on-line application. Even if a total rewrite is required, this can be easily accomplished in the 10 man-year time constraint. This is the only step in the migration plan that is difficult to justify on financial grounds. We saw no way to avoid violating the ground rules for this particular step.

At the end of the sixth migration, the account balances remain in the VSAM system and only about 72K lines of *kernel* VSAM update code actually access this data. This remaining database must be migrated as the last step.

Migration 7: Move the account data from VSAM to SQL

This final step is trivial to justify, because it would allow the mainframe to be retired, resulting in a large hardware savings.

3.4 CMS Cut-over Issues

Each step of the above migration plan introduces serious cut-over problems. It is impossible for CMS to be *down* for more than a few hours. If the new system fails to work for any reason, the old system must resume operation immediately. Any failure to accomplish a smooth transition would be a cause for immediate dismissal of the bank personnel involved.

Cut-overs that entail rewritten applications are straightforward. Beginning with the old system, one can migrate users, one by one. Hence, at any given point in the cut-over process, some fraction is running the legacy application while the rest run the corresponding target application. IS personnel can move users at whatever rate is comfortable. Recovery from a catastrophe can be accomplished by restoring the old application code.

The database migration and cut-over steps proposed for decomposable ISs (Steps D7 and D10) pose serious problems for CMS. There is considerable overhead in using a forward gateway to map DBMS calls in the old system to SQL and then a reverse gateway to map them back to the old DBMS. This would seriously tax the IBM machine currently running CMS. In addition, distributed DBMS software with these features was not yet widely available or robust enough for the bank's requirements.

Because of these problems, we proposed an alternate approach for database migration and cut-over. To migrate any portion, P, of the database, replicate P in the target database (i.e., have P appear in both the legacy and target databases). Identify all transactions that update P and execute them twice; once on the legacy database and once on the target. Depending on whether they originate in 4GL code or COBOL, they will execute directly against the database or use one of the two gateways described in Section 2.

In essence, the target database is brought up in parallel with the legacy database. For a while, duplicate transactions are performed. Subsequently, P would be removed from the

legacy database and the duplication of transactions would be turned off. The cut-over of P is now complete. The replication of P does not require a distributed DBMS or the cascaded use of a gateway. Hence, it will require less overhead on a crowded mainframe, and therefore it was chosen as the better alternative. However, the gateway must implement update consistency.

3.5 CMS Migration Summary

The CMS migration plan consists of two major steps:

- Incrementally migrate the legacy applications.

CMS is a decomposable IS. Modules can be *peeled off* one by one and moved to a new environment in manageable sized *chunks*.

- Incrementally migrate the legacy database.

The database can be divided into multiple pieces, which can be migrated one by one. Although cut-over is a problem, it can be accomplished using the *brute-force* method of running the two databases in parallel.

After completing these two steps, we were left with a small *core* that had manageable complexity and could be moved as the final step. It is our assertion that most complex ISs can be *peeled* in this fashion. If CMS's architecture had been poorly designed (e.g., if Xcash had performed its own updates instead of calling Xcheck) then the *core* would have been larger. In this case, we speculate that the re-engineering of multiple kernels into a single kernel would have been the appropriate step (e.g., in Step D3).

Lastly, note that our migration plan is an incremental rewrite of CMS and not incremental re-engineering. Although there has been much interest expressed in re-engineering legacy ISs, our case study has indicated that virtually all code would be better rewritten using modern tools, especially 4GLs, report writers, and relational DBMS query languages. There may be legacy ISs for which re-engineering is a larger part of the migration plan. However, our experience with CMS did not indicate any significant use of this technique.

4. MIGRATING SEMI-DECOMPOSABLE LEGACY ISs

This section presents a migration method for semi-decomposable legacy ISs. The migration architecture for a semi-decomposable legacy IS contains an application gateway placed between the interfaces and the legacy and target ISs, as illustrated in Figure 4.1. The application gateway maps application calls from the interface modules to application calls in the legacy and target applications. This method applies the reverse and forward migration methods, described in the previous section, at the application call level.

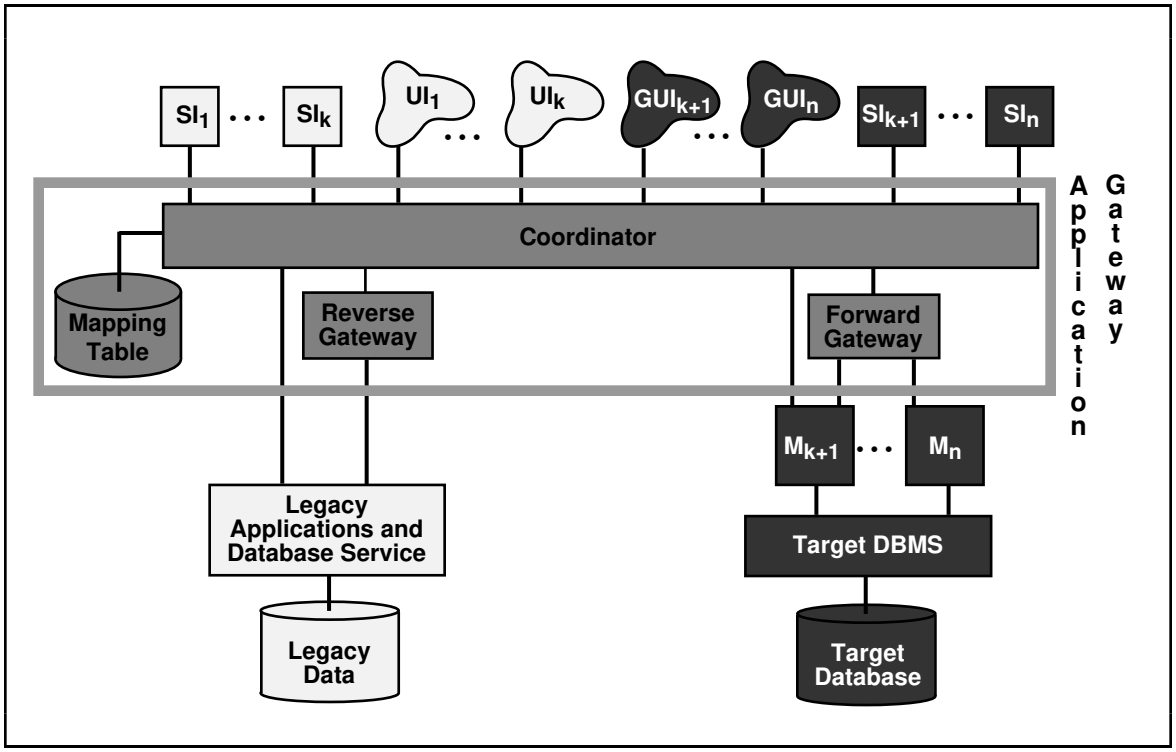


Figure 4.1: Semi-Decomposable Legacy IS Migration Architecture

4.1 Migration Method For Semi-decomposable Legacy ISs

The migration method for semi-decomposable legacy ISs starts with a semi-decomposable legacy IS, illustrated in Figure 1.2, and produces a target decomposable IS, illustrated in Figure 2.1. We comment here only on extensions to the migration method for decomposable legacy ISs. We also separate interface design from application design. This could be warranted based on the complexity of the legacy and target ISs. This separation was deferred to this section to simplify the presentation of earlier methods.

Step S(emi-decomposable)1 Iteratively install the target environment. (See Step D1)

Step S2 Iteratively analyze the legacy IS. (See Step D2)

This step is more complex than for decomposable legacy ISs due to the increased complexity of the architecture.

Step S3 Iteratively decompose the legacy IS structure. (See Step D3)

Improve the legacy IS structure by eliminating as many dependencies as possible between applications and between the applications and the database service. This will simplify the migration that otherwise must support all the dependencies. This step may be too complex or may introduce too much risk. In the worst case, the legacy IS will remain in its original form. This will complicate the gateway, increase costs, reduce performance, and add risk.

Step S4 Iteratively design the target interfaces. (See Step D4)

Design the end user interfaces and system interfaces for the target IS. Develop an interface migration plan from the legacy, through the composite, to the target IS. To ease the migration of end users, it may be helpful to use an interface gateway since it provides a single composite IS interface that supports both legacy and target interfaces, and assists with the migration between them.

Step S5 Iteratively design the target applications. (See Step D4)

Design the target applications based on requirements that result from the analysis (Step S2). Additional requirements or target functions increase the risk of failure.

Step S6 Iteratively design the target database. (See Step D5)

The less knowledge there is about the database or application structure, the more likely it is that the legacy and target databases will have to run in parallel. This increases the importance and difficulty of coordinating the databases and defining and maintaining the correctness criteria for the composite, operational IS.

Step S7 Iteratively create and install the application gateway. (See Step D6)

This may be the most technically challenging step. It should be left to appropriately skilled experts. The application gateway captures legacy and target application calls from legacy and target interfaces and either passes them unchanged to the legacy and target applications (respectively), or translates them to target and legacy application calls (respectively). It must capture the corresponding results, translate them if needed, and direct them back to the appropriate interfaces. The application gateway must coordinate target and legacy updates when the two simultaneously support duplicate applications or data. Application coordination could be similar to but more complex than database coordination described for the database gateway in Step D6.

Step S8 Iteratively migrate the database. (See Step D7)

It may be useful to augment the gateway to assist with database migration since the required mapping information may be in the gateway's migration database.

Step S9 Iteratively migrate the legacy applications. (See Step D8)

Step S10 Iteratively migrate the legacy interfaces. (See Step D9)

Step S11 Iteratively cut-over the target IS. (See Step D10)

The lack of structure in the legacy applications and database service may make it difficult to determine if a legacy component can be retired when the corresponding target component has been cut-over.

5. CASE STUDY 2 MIGRATING TPS

One of the authors participated in developing a migration plan for the Telephone Provisioning System (TPS) for a large telecommunications company. This was an opportunity to apply the migration method for semi-decomposable legacy ISs and extend it for the non-decomposable case. This section describes the semi-decomposable legacy IS, TPS, and the proposed migration plan.

5.1 TPS

TPS supports aspects of telephone provisioning. Telephone provisioning involves allocating and deploying telephony resources to provide a customer with a requested telephone service. Assume that a customer has called the telephone company to request telephone service. The telephone company must perform the following tasks:

- Verify the street address.
- Identify an available telephone number and the required equipment (e.g., cable-pairs, line equipment, location on a distributing wire frame, jumper wires, special circuits for advanced services, cross connect box) from its available inventory.
- Assign them to the customer and take them out of inventory, deploy them (i.e., make the necessary connections, update the telephone switch that will serve the customer).
- Ensure that it works.
- Inaugurate the service.

Many related functions are initiated directly or indirectly by TPS, including customer credit validation, account and billing set up, and directory services update. Ideally, service provisioning is completed during customer contact.

This process is required for any change to a customer's telephone service. It is invoked over 100,000 times per day for the current customer base of 20,000,000. If one process fails, a customer may lose the requested telephone service. If many fail or if TPS goes down, large numbers of customers can be affected within minutes. Other types of failure can be costly. If inventory or assignments are not done correctly, equipment may be kept out of service or may be inconsistently assigned, resulting in poor service or no service at all. If billing is not initiated correctly, the company may not be able to bill for delivered services or may bill for services not rendered. TPS is clearly mission critical.

TPS supports four basic functional areas:

- Core functions: cable-pair assignment, inventory management, wire frame management, switch changes, network planning, central office conversion support, repair support, performance audits, and related reporting and auditing.
- Street address guide (SAG): manages all valid addresses, counts services provided at each address, validates customer addresses, and identifies location for equipment assignment.

- Dial office administration (DOA): assigning resources that optimally meet the customer and company requirements. DOA functions include assignment of telephone numbers and line equipment, load balancing of switching and line equipment, managing equipment reservations and availabilities, aging telephone numbers (one doesn't want the one the local pizza parlor just gave up), equipment rotation, and related reporting and administration.

- Outside plant management (OSP): managing and assigning telecommunications equipment (e.g., serving terminals and cross connect boxes) that is not inside a central office.

TPS's key resource is the data it uses to run and administer telephone provisioning. Over forty major ISs require information from, and actions to be performed by TPS. For example, service order entry systems are the source of most telephone provisioning requests to TPS. Other major ISs requiring on-line access to TPS include systems for equipment testing, billing, verification, analysis, reporting, trouble shooting, repair support, telecommunication network management, and equipment ordering. Over 1,200 small systems (e.g., report writers) also require access to TPS's data. Hence, a fifth functional area is the provision of interfaces to many other ISs.

TPS is currently just under 10^6 lines of FORTRAN code running on a Honeywell mainframe under a proprietary operating system. Its implementation consists of a massive program library (i.e., 6,000 source code files, 1,500 executable modules) and an immense *database* (i.e., over 10^6 data files of 900 file types totaling approximately 400 gigabytes of data). TPS's transaction rate (10^5 per day) grows significantly during special operations (e.g., central office conversions). Data volumes, processing, and accesses grow at 20% per year. Considering the functions it was designed to perform, TPS has run remarkably well and reliably, 24 hours a day, for twenty years with high user satisfaction. This operational reliability is one if its best features. As with most such legacy ISs, there are several development versions and several operational versions (due in part to regulatory differences in deployment regions). This complicates the migration since the versions must all be maintained.

The history and problems of TPS are typical of large legacy ISs. TPS began its life, in 1972, as a small IS for managing data for installation and dispatch administration (e.g., installation and repair support) and switching services (e.g., switch changes, wire frame management). At that time, TPS was ideal for managing data for the then current telephony technology and business processes and rules. As telephony and related business evolved, TPS was extended or altered to support these unanticipated, mission critical changes. Hence, there was seldom a global plan with which to control the evolution or against which to develop a comprehensive, integrated design. Over its twenty year history, it grew by a few large extensions and a vast number of enhancements, modifications, and fixes. TPS's inflexible and complex structure reflects this history. Applications are neither modular nor cleanly separable. Files, data structures, and indexes are convoluted, complex, and poorly structured. Programs and data are highly interdependent and redundant. This makes TPS hard to modify and renders some applications and data outdated or irrelevant. Under current conditions, it is difficult to allocate resources or time to fix known problems (e.g., pointer chain failures that cause crashes and are costly to fix), let alone to make changes necessary to meet some current and future requirements. To complicate matters, other ISs have been built to avoid modifying TPS but require interactions with TPS.

The above concerns for this mission critical system motivated over 25 studies to replace TPS. The failure of these studies to develop a feasible plan and the failure of at least one

Cold Turkey effort, led the company to request the study described here.

5.2 TPS Analysis and Migration Challenges

This sub-section presents an analysis of TPS, identifies potential problems, and where possible, proposes means of reducing them in the migration plan.

Analysis was complicated since there is no complete specification or documentation. What documentation exists is outdated since changes are made so rapidly and so often that the requirements, specification, and documentation have not been kept current. The system itself is the only complete description.

As with the CMS case study, it was previously believed that TPS was hard, if not impossible, to decompose. Our analysis suggests that it could be decomposed into eleven logical components. As illustrated in Figure 5.1, there are four major functional components (Core TPS, SAG, TPS DOA, TPS OSP), two minor functional components (advanced services administration and validation), three interface components for major ISs (TPS Service Order Entry (SOE), TPS Interface Application (MIA), and TPS Switch Interface (MSI)), one interface component for minor systems (TPS Access Interface), TPS utilities (not illustrated), and a database services component that manages TPS data.

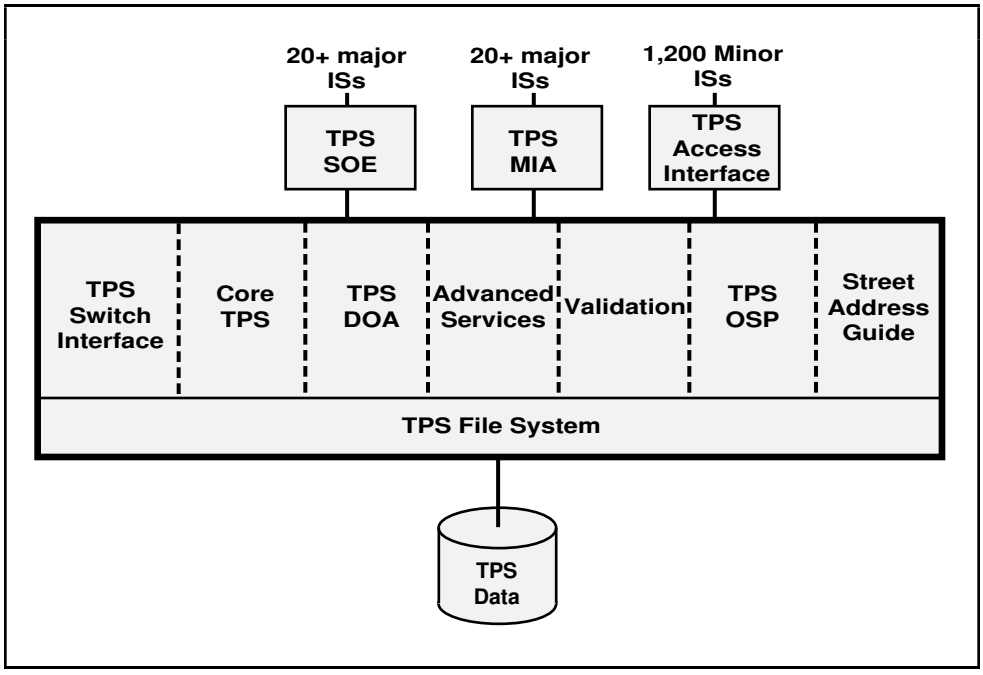


Figure 5.1: TPS Architecture

Through analysis, we also found that the TPS data could be decomposed into seven logical databases; one each for SAG, OSP, telephone numbers, pending orders, cable-pairs, wire frames, and inventory (e.g., line equipment and miscellaneous equipment). A large number of files were found to be artifacts of the implementation (e.g., indexes, output and administrative files) that need not be replicated in the target IS.

TPS's critical resource is the data, not the functionality. Hence, the primary goal of the TPS migration was determined to be data liberation (i.e., migrating TPS data to a modern

DBMS accessible by other ISs). The target database must be designed to meet the current and known future requirements and be flexible enough to facilitate change. The lack of knowledge of the current data structures was not considered a major database design problem, since the database would not be duplicated, rather its logical requirements would be met. The logical requirements could be derived from the code, the actual data, the functions, the current and planned operation, and from TPS experts. Since many known problems, including redundancy, are artifacts of the legacy design and implementation, they can be ignored when designing the target database which will be correspondingly smaller with redundancies removed. A wisely chosen distributed target DBMS could meet all known requirements, including those for performance.

The major database challenges concern database cut-over, data migration, and copy consistency. Database cut-over is a challenge due to the sheer volume of TPS data and to the requirement for non-stop operation. Migrating TPS data to the target database is complicated by the lack of knowledge of the data structures and the limitations of TPS's database services. Following the Chicken Little strategy, we felt that the solution involved partitioning TPS data into small enough chunks for separate migrations, and sequencing the migrations. Copy consistency requires that updates to TPS data be reflected in the corresponding copy in the target database. Defining the corresponding correctness criteria for coordinating updates is hard and only part of the problem of establishing a mapping between the two databases. Supporting the required distributed transactions in the gateway is hard, requiring expert database systems skills. Both of these problems could be significantly reduced by building the gateway using a distributed relational DBMS.

Through analysis, we found that a significant amount of TPS interface code was redundant or obsolete and need not be migrated. The IS and user interfaces were redundant and in need of major improvements. We recommended that all user interfaces be migrated from the current TTY technology to PC-based GUIs. The user interfaces significantly affect many clerks. The IS interfaces significantly affect over 40 existing mission critical ISs and possibly many future ISs. Management was rightly concerned about both of these aspects. Hence, the benefits of interface improvements were used to sell the interface migration to management. Two benefits were interface standardization and the "single point of contact" policy intended to permit clerks to access any information that might be required during the customer contact.

As with the CMS migration, the user interfaces can be rewritten efficiently using the 4GL of a relational DBMS. Due to the size of the interface migration, we recommended using standard user interface development environments and tool kits (e.g., Windows, X, Motif, Openlook).

In analysis we found that the scope of application migration was less than anticipated. Many TPS functions were found to be redundant, obsolete, and inefficient or would soon be so due to modern telephony equipment. Hence, many applications need not be migrated at all. Those that should be migrated should be correspondingly modified and simplified. As with the CMS migration, we reduced the application migration problem by focusing on only those functions that needed to be migrated.

We did not have a good picture of the TPS workload or of what functions and data were most frequently accessed. We recommended that TPS be instrumented to capture this information as an aid in designing the target TPS.

The TPS migration gateway was considered to be the greatest technical challenge. As discussed in Section 4, such a gateway is hard to design, evolve, and maintain throughout the migration. TPS's performance and interface requirements potentially add significantly

to the challenge. We found ways to reduce the problem. First, up to 80% of all TPS accesses are read only. When data has been migrated to the target DBMS, all read-only access can be directed there. This will improve performance, simplify the gateway, and reduce the load on TPS. The copy consistency problem is reduced since not all TPS functions and data will be migrated. Also, since only 20% of TPS accesses involve updates, function migration and cut-over chunks should be selected for migration and the migrations sequenced to further reduce the copy consistency problem. Due to TPS's complex structure, it is hard to determine when a migrated function can be retired, since live TPS functions may still depend on it.

Early database migration permits benefits discussed above and provides a base for advanced provisioning functions not feasible under TPS. This could be sold to management based on the related new revenues for the services that could not be provisioned using TPS. Hence, we recommended that the database migration be accelerated. Function migration should be planned and sold to management based on reducing TPS's running and maintenance costs and on simplifying the gateway.

Most problems identified in the analysis can be reduced by selecting appropriately sized chunks to migrate and an appropriate sequence for the migrations and cut-overs. Initially, chunks should be very small and independent. When the TPS migration is well underway and the risks and solutions are better understood, larger chucks may be reasonable.

The seven logical data chunks and corresponding functions, mentioned in Section 5.2, are too big for single migration steps given the operational requirements (i.e., migration time exceeds available down time). Smaller chunks had to be found. The telephone network is supported by over 4,500 central offices each of which contains one or more telephone switches and can be treated independently. They are natural units of migration in the telephone business. Central offices are the traditional units of conversion when upgrading the telephone network (e.g., upgrade or convert switches). Indeed, TPS itself is designed to facilitate central office conversion. We proposed a sequence of migrations based on chunks of TPS functionality, starting with SAG, and portions of TPS data for one central office. Once a function or data chunk is migrated for one central office, the cut-over can start in more central offices.

5.3 The TPS Migration Plan, Part I

This sub-section outlines the first few migrations of the proposed plan. Each migration is intended to follow the method for semi-decomposable ISs. Due to the mission critical role of TPS and its unsuccessful migration history, TPS management requested a migration plan for these first few steps as a basis to study and verify the method. We estimated that it could take up to eight years to complete a full TPS migration. During that time a detailed plan developed today would become obsolete. Hence, it was agreed that these first migrations be considered pilot studies or experiments to better understand the problems, technologies, and skill requirements as a basis for subsequent migrations.

We found that three portions of TPS, SAG, OSP, and the TPS Switch Interface, were relatively easy to separate, migrate, and retire after migration. Hence, these are the focus of the first three migrations.

TPS portions other than SAG, OSP, and the TPS Switch Interface, are not easily separable. Hence, they are more complex to migrate and cut-over. It may not be possible to retire the remaining legacy portions once the target portion is cut-over since there may still be dependencies between the various legacy portions that prevent this. Accordingly, the gateway must be designed to coordinate updates between legacy portions and their target

counterparts. All non-update accesses can go to the target portions. Since portions of TPS may be important enough to keep but not important enough, or too complex, to migrate, those legacy portions may remain in the ultimate TPS. The corresponding gateway functions may also be required.

Figure 5.2 illustrates the TPS migration architecture that would result from the first migration. It includes TPS, as illustrated in Figure 5.1, and a decomposable target IS. The TPS application gateway manages all user and IS accesses since it encapsulates the legacy and target versions of TPS. The database component of the gateway will grow and shrink as the legacy database is iteratively migrated to the target DBMS and the composite IS evolves. We recommended a distributed DBMS, due to target IS distribution requirements.

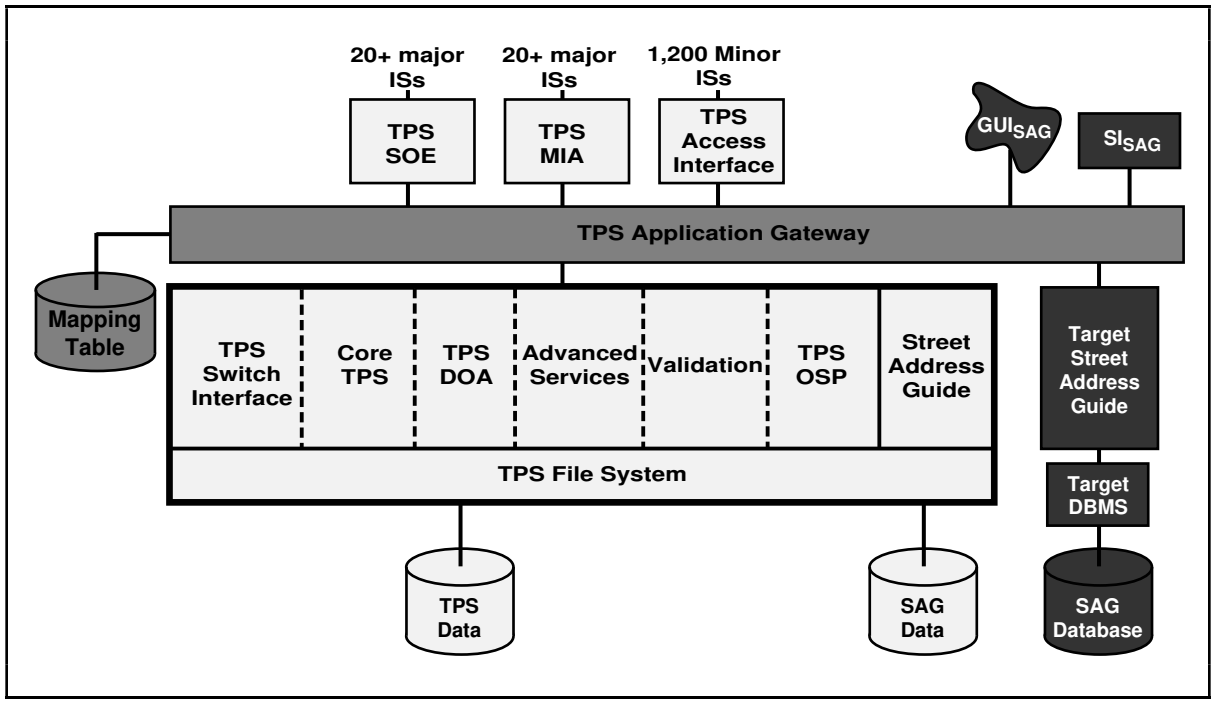


Figure 5.2: TPS Migration 1 Architecture

Migration 1: Migrate The Street Address Guide.

We proposed for the first migration and pilot study that the SAG database and associated functionality and interfaces be migrated. This was because SAG is the most easily separable portion and because SAG is required in the target TPS. The SAG migration could proceed as follows. Decompose TPS by separating the Street Address Guide (SAG) application module from the other application modules and the SAG data from the legacy data, as illustrated in Figure 5.2. Create and install the TPS Application Gateway to deal only with SAG calls. Migrate the SAG interfaces, application, and database. We recommended that the migration and cut-over be done iteratively, each iteration focused on one or more central offices, involving less than 100 megabytes of data. If successful, the iterative cut-over could proceed with increasingly larger numbers of central offices until all 4,500 have been migrated. Once SAG is cut-over, retire the legacy SAG application and data from legacy TPS.

Migration 2: Migrate Outside Plant.

We proposed OSP for the second migration, since it was the next easiest to separate. However, the SAG and OSP migrations could proceed in parallel since they are so similar. The OSP migration proceeds as follows. Separate the legacy OSP application module and data from TPS, as illustrated in Figure 5.3. Augment the TPS Application Gateway to deal with OSP calls. Migrate the OSP interfaces, application, and database. We recommended an iterative cut-over again focused on central offices. Once OSP is cut-over, retire the legacy OSP application and data from legacy TPS, as illustrated in Figure 5.4.

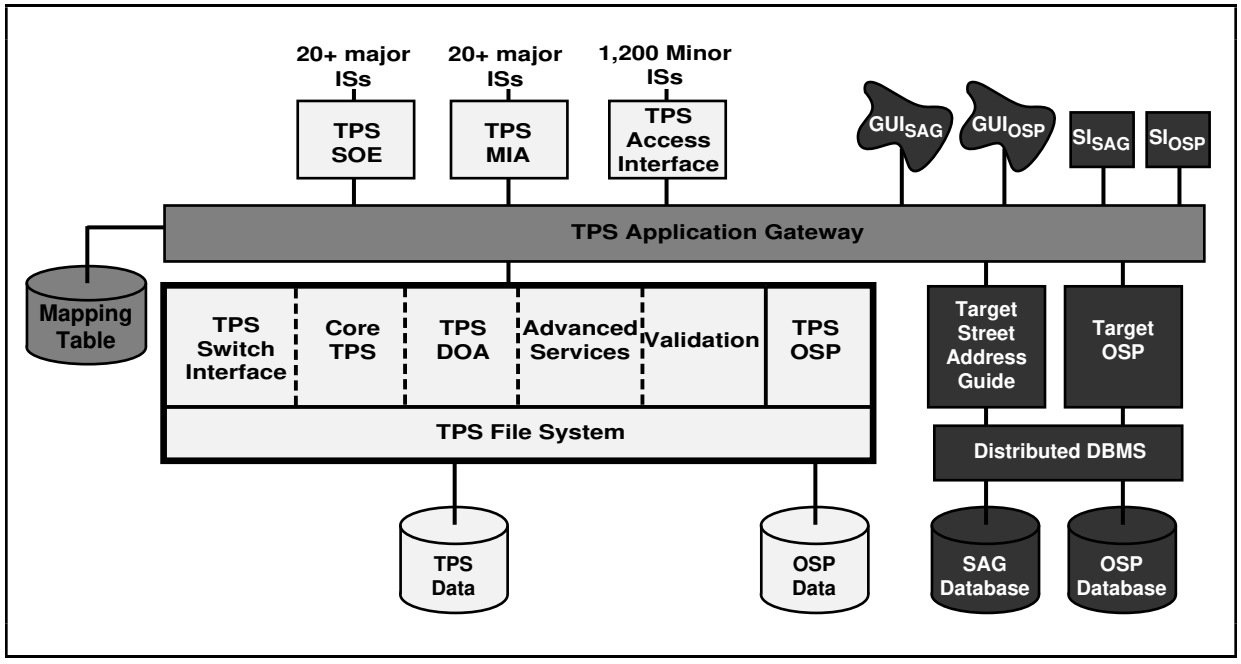


Figure 5.3: TPS Migration 2 Architecture

Migration 3: Migrate TPS Switch Interface.

For many of the reasons cited above, we recommended TPS Switch Interface for the third migration. It proceeds as does SAG and OSP and has the migration architecture illustrated in Figure 5.4.

Migration 4: Migrate The TPS Interfaces.

The TPS interface migration is a large, complex process. It is mission critical since all interfaces must remain operational throughout the migration. This includes interfaces for over 1,200 minor ISs and 40 major ISs. The TPS interface migration was complicated by a second factor. Further analysis revealed that the applications yet to be migrated were not decomposable, as were some of their user and system interfaces. To simplify the migration, we proposed that the non-separable interfaces be eliminated by altering the legacy applications to use one of the separable TPS interfaces (i.e., TPS SOE, TPS MIA, TPS Access Interface). we viewed this as necessary even though it violated our goal of altering legacy code as little as possible. As a result, there will be a large number of legacy user and system interfaces to be migrated.

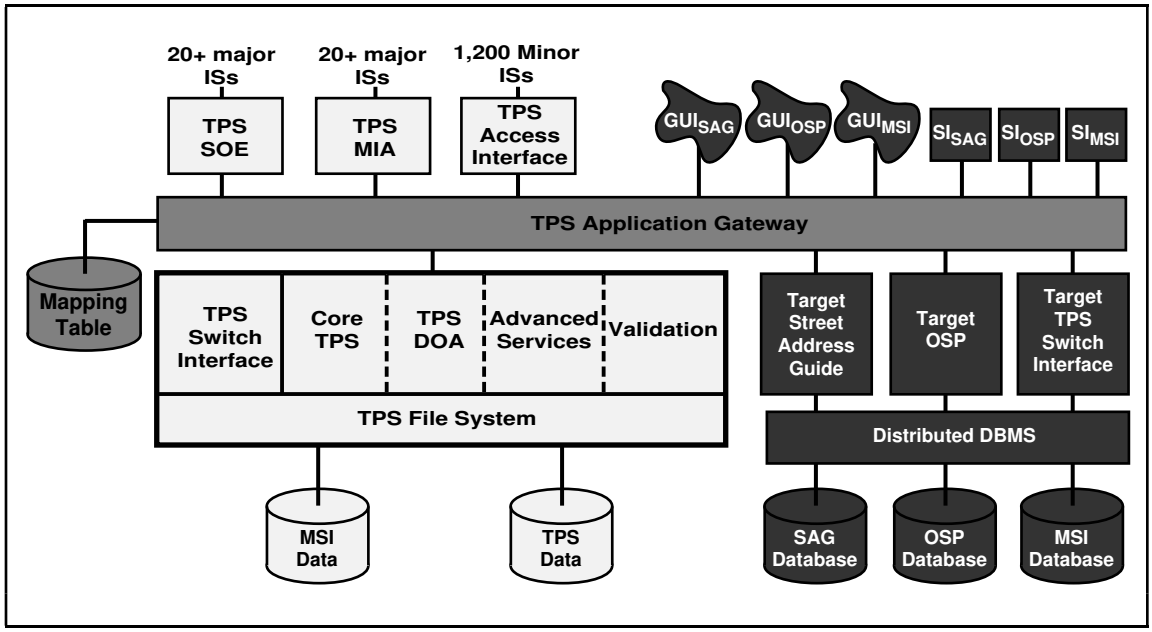


Figure 5.4: TPS Migration 3 Architecture

We proposed that the TPS interface migration be done using an interface gateway, illustrated in Figure 5.5. The TPS interface gateway provides the appearance to all existing ISs and end users that they continue to access the same interfaces. However, the gateway can redirect the calls and the results appropriately to the current state of the migration. The organization decided to use the interface gateway to introduce corporate interface standards and temporary versions of critical features that would not otherwise be in place until much later. Under cover of the TPS interface gateway, the legacy IS interfaces (TPS SOE, TPS MIA, and TPS ACCESS Interface) must be migrated to the appropriate target IS interfaces. We decided to develop one uniform IS interface to replace all three legacy IS interfaces.

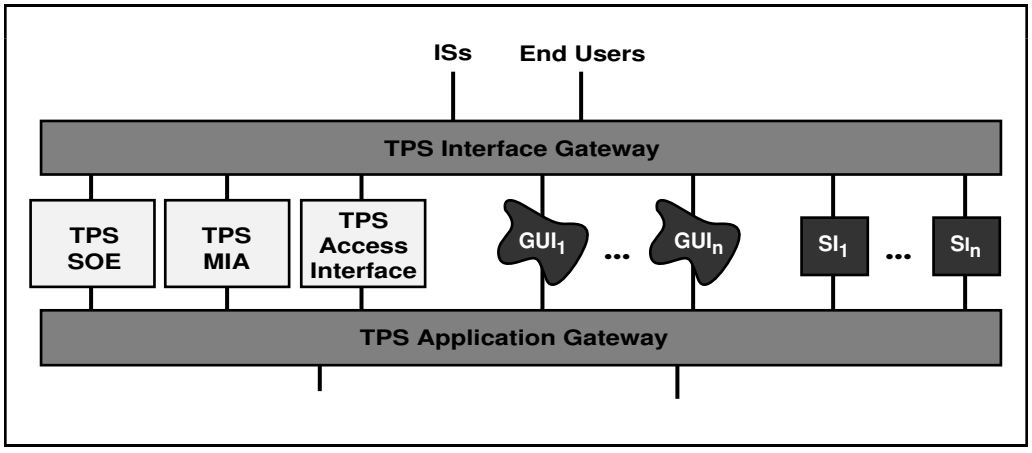


Figure 5.5: TPS Interface Gateway

Although TPS was believed to be semi-decomposable, it was found to be hybrid, as

illustrated in Figure 1.4. That is, after migration 3, we found that the remaining applications were non-decomposable, except by significant changes to the legacy IS which were deemed to be far too risky. Hence, migration 4 ended the semi-decomposable migration. The remainder of TPS required a migration method for non-decomposable ISs as described in the next section.

6. MIGRATING NON-DECOMPOSABLE LEGACY ISs

This section presents a migration method for non-decomposable legacy ISs, illustrated in Figure 1.3. This migration method is an extension to the migration method for semi-decomposable legacy ISs, described in Section 4. Our starting point is a non-decomposable legacy IS, illustrated in Figure 1.3. Our target is a decomposable legacy IS, illustrated in Figure 2.1.The migration architecture for non-decomposable legacy ISs, illustrated in Figure 6.1, includes an IS gateway that combines the non-decomposable legacy IS and the decomposable target IS to form a composite IS. We focus on the major differences required to deal with ISs that lack structure and on the key challenges which concern analysis and partitioning.

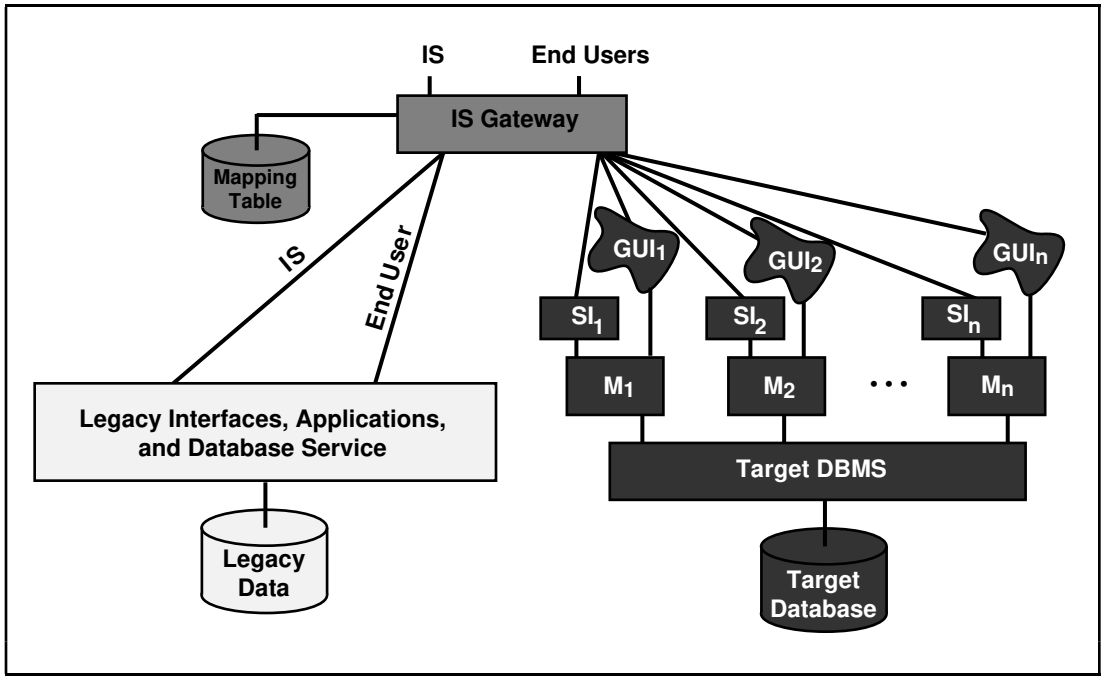


Figure 6.1: Non-Decomposable Legacy IS Migration Architecture

6.1 Migration Method For Non-decomposable Legacy ISs

Step N(on-decomposable)1 Iteratively install the target environment. (See Step S1)

Step N2 Iteratively analyze the legacy IS. (See Step S2)

This step can be arbitrarily complex. In the worst case, the legacy IS must be treated like a blackbox. Its functions and data content must be analyzed from whatever information is available, such as documentation, people, dumps, the history of its operation and services provided. It may be useful to conduct experiments to probe the system using the known interfaces and available tools.

The lack of structure may mean that the requirements must be defined from scratch (i.e., distinct applications or functions may not be clear). When appropriately small chunks of

data or function can be identified but not proven to be independent, they may simply have to be replicated and the legacy and target copies coordinated until it is demonstrably safe to retire the legacy versions.

Step N3 Iteratively decompose the legacy IS structure. (See Step S3)

"Peeling the onion" by taking off separable portions may produce a number of decomposable components and a non-decomposable core. These can be treated separately. Even if non-decomposable legacy ISs cannot be decomposed, some restructuring could facilitate the migration.

Step N4 Iteratively design the target interfaces. (See Step S4)

Step N5 Iteratively design the target applications. (See Step S5)

Step N6 Iteratively design the target database. (See Step S6)

Step N7 Iteratively create and install the IS gateway. (See Step S7)

An IS gateway encapsulates an entire IS, as illustrated in Figures 1.6 and 6.1. It is similar to an interface gateway except that it must provide the coordination function with the information available at the user and system interface levels. This makes the IS gateway very difficult to build. It could also include a *communications gateway* required to migrate from one communications technology to another (e.g., IBM's CICS to TCP/IP), thereby assisting the migration from mainframe to client-server architectures.

In previous methods, the gateway functionality decreases as legacy IS components are retired. Due to the lack of structure in the non-decomposable legacy IS, it may not be possible to retire any legacy components until the migration is complete. Hence, the IS gateway will continue to increase in complexity until the non-decomposable legacy IS can be retired. Further, it may not be useful to migrate some legacy IS components at all even though they are still required. For example, they may be required for seldom accessed, archival information or functions. Hence, the IS gateway may become an integral part of the ultimate IS.

Step N8 Iteratively migrate the legacy database. (See Step S8)

The difficulty of this step depends on the results of the previous steps. It may be very difficult or costly to access data in legacy database services in non-decomposable legacy ISs. This may be due to the legacy database service, the applications, or the structure of the data. All of these factors applied to the TPS migration, which was a particularly bad case of non-decomposable legacy database migration. First, there was almost no knowledge of the internal structure of the legacy database. This was complicated by the fact that the physical structure was cluttered with implementation artifacts that were hard to distinguish from application data. As a result, we proposed that the existing legacy applications (e.g., database query, report generation, and access routines) be used to extract legacy data. This significantly lengthened the database migration time.

Step N9 Iteratively migrate the legacy applications. (See Step S9)

Step N10 Iteratively migrate the legacy interfaces. (See Step S10)

Step N11 Iteratively cut-over the target IS. (See Step S11)

7. MIGRATING LEGACY ISs (GENERAL CASE)

In this section, we discuss the general case migration method for legacy ISs. The migration method is a combination of the previous migration methods. This method is the most broadly applicable of those presented in this paper. It applies to all legacy ISs since each legacy IS is a special case of the general case illustrated in Figure 1.4. In general, a legacy IS may have a portion that is decomposable, a portion that is semi-decomposable, and a portion that is non-decomposable. The migration architecture for this general case is a combination of the migration architectures for decomposable, semi-decomposable and non-decomposable legacy ISs, as illustrated in Figure 7.1. A simple combination of the gateways, illustrated in Figure 7.1 may not be appropriate. For example, it may not make sense to place the target application modules M_{l+1} through M_n above the database gateway. Alternatively, they could be placed below the gateways directly accessing the modern DBMS. In this case, the corresponding GUI's (GUI_{l+1} through GUI_n) would interact directly with the application gateway. In any case, the migration architecture must be tailored to the specific requirements of the legacy IS to be migrated.

We now outline the migration method for arbitrary legacy ISs. It is a combination of the migration methods for decomposable, semi-decomposable, and non-decomposable legacy ISs. Our starting point is a legacy IS, illustrated in Figure 1.4. Our target is a decomposable legacy IS, illustrated in Figure 2.1.

Step L(egacy IS)1 Iteratively install the target environment. (See Steps D1, S1, N1)

Step L2 Iteratively analyze the legacy IS. (See Steps D2, S2, N2)

Identify the portions of the legacy IS that are decomposable, semi-decomposable, and non-decomposable.

Step L3 Iteratively decompose the legacy IS structure. (See Step D3, S3, N3)

Decompose the legacy IS into portions that are decomposable, semi-decomposable, and non-decomposable.

Step L4 Iteratively migrate the portions identified in Step L3.

Apply to each portion identified in Step L3 the appropriate legacy IS migration method. As with the previous methods, it is a significant challenge to coordinate the individual migrations.

7.1 The TPS Migration Plan, Part II

The TPS migration, described in Section 5, became an example of the general case when we realized that, for all practical purposes, some applications were non-decomposable. This understanding led to the following TPS migration plan.

Step TPS L1: Iteratively install the target TPS environment.

Step TPS L2: Iteratively analyze TPS.

This analysis, described above, determined that TPS could be decomposed into: three separable applications TPS SAG, TPS OSP, and TPS Switch Interface; three separable IS

interfaces, TPS SOE, TPS MIA, and TPS ACCESS Interface; and the rest of TPS, called TPS-rest. Further analysis found that part of TPS-rest was semi-decomposable, illustrated as TPS-rest-S in Figure 7.2. The remainder of TPS-rest, TPS-rest-N was found to be non-decomposable.

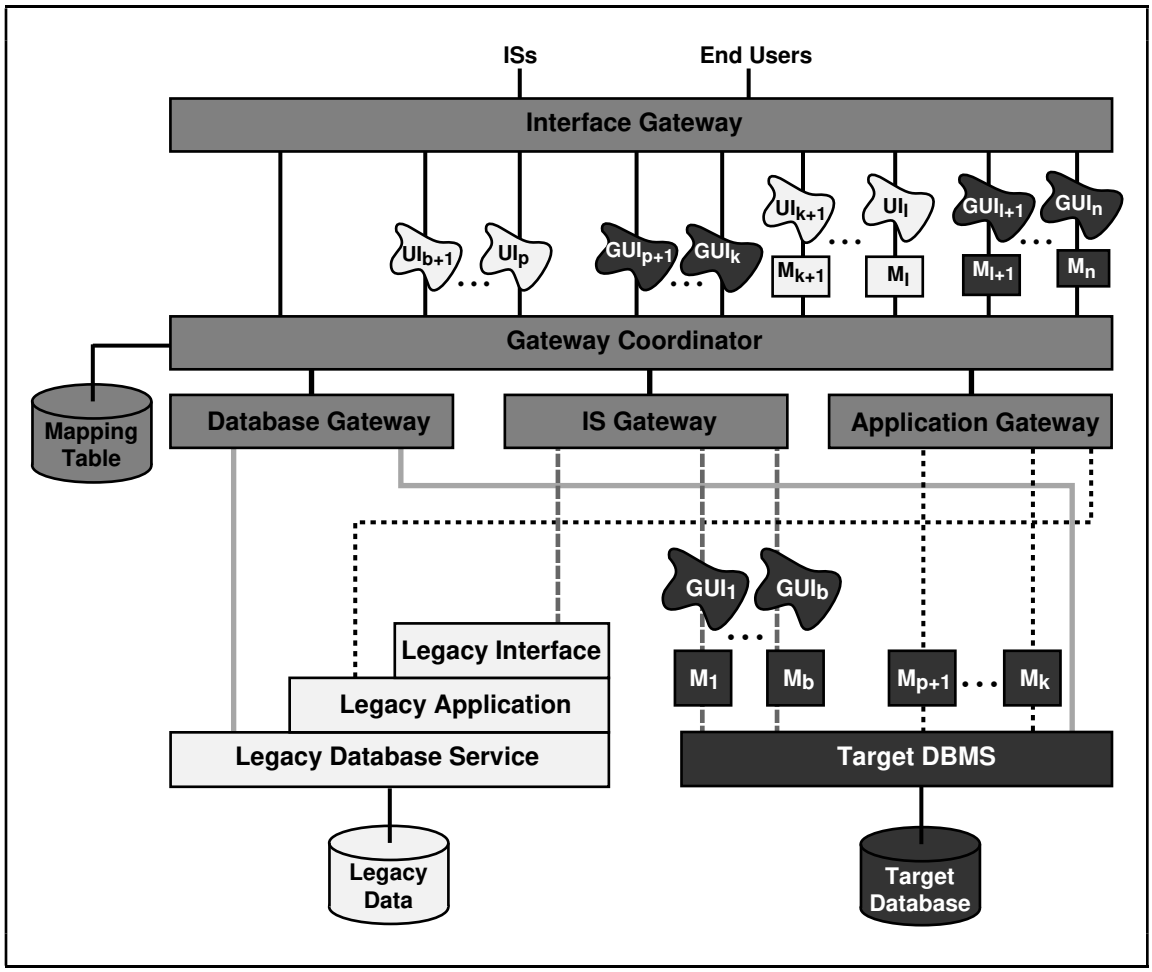


Figure 7.1: Hybrid Legacy IS Migration Architecture

Step TPS L3: Iteratively decompose and migrate TPS.

Apply the decomposable legacy IS migration method to the decomposable IS components, as is proposed in Section 5.

Apply the interface migration method to the decomposable IS interfaces, also described in Section 5.

Apply the non-decomposable legacy IS migration method to TPS-REST-N.

The resulting TPS migration architecture is as illustrated in Figure 7.2

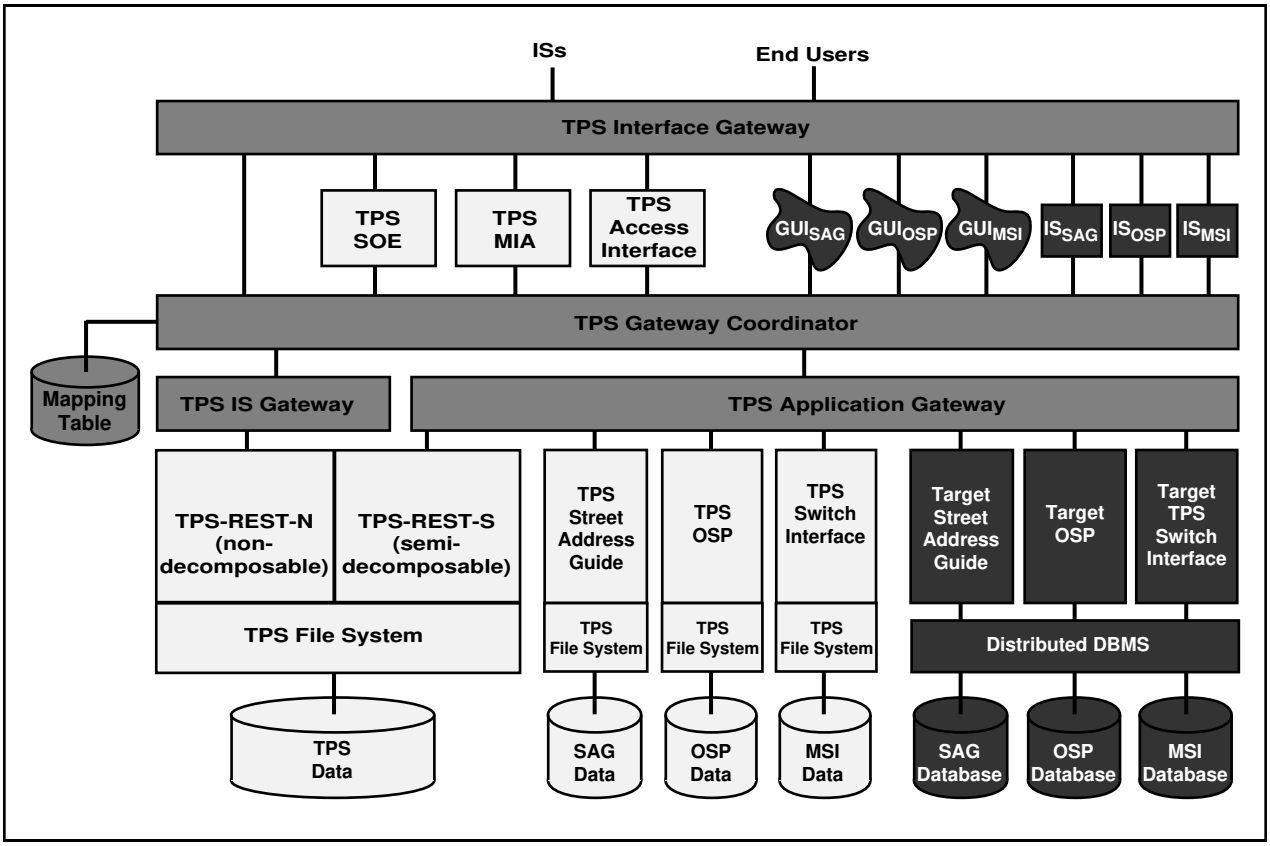


Figure 7.2: Worst-Case TPS Migration Architecture

8. RESEARCH AGENDA

There are few concepts and almost no adequate tools or techniques to support legacy IS migration. Those that exist do not scale up to meet the challenges posed in the design, development, maintenance, or evolution of large scale ISs [BROD92]. This section comments briefly on technologies and tools that could dramatically help the migration methods described above. Many of these suggestions require research.

8.1 Gateways

Gateways are critical to all migration methods and architectures proposed in this paper. They also pose the greatest technical challenges and can be extremely costly to build and operate. Indeed, we recommended that they be designed and built by database systems experts. It would be enormously valuable to develop generic gateway technology to encapsulate databases, interfaces, communications, or any other major system component. Gateways are sometimes called *surround* technology. For example, CMS needs to move from VSAM files to DBMS technology *seamlessly* (i.e., the old must gracefully evolve to the new without a discontinuous break).

Gateways are much more broadly applicable than simply for migration architectures. They may become major components of future IS architectures. Whenever two systems are required to interact, some interface is required. If the interactions involve anything more than simple remote procedure calls, it is often necessary to develop a software interface component. Such a component is a gateway (i.e., between the two systems). Gateways have stringent requirements. For example, those between mission critical ISs must be very reliable, robust, and efficient. They cannot impose a significant performance penalty on the old IS (say more than 10 percent). It is not clear how to achieve this level of performance.

Requirements for IS interaction (often misleadingly called IS integration) and distributed computing (e.g., client-server) are growing dramatically. Gateways are a frequently proposed solution. Indeed, some proposals for next generation computing [OMG91a, OMG91b, MANO92, BROD92, BROD93] are based on gateways. Research is required to understand and develop generic gateway technology. There is an increasing number of ad hoc (i.e., for a small number of specific systems) gateway products currently with minimal functionality (e.g., no update coordination). General purpose tools are required for constructing gateways, just as DBMSs are generic tools for constructing data intensive applications. Ad hoc gateways are extremely complex and costly to construct. This can be seen in the cost and complexity of gateways provided by DBMS vendors to support heterogeneous distributed DBMSs, focused on their product.

8.2 Specification Extractor

Specifications for legacy ISs are nearly non-existent. For both CMS and TPS, the code itself is almost the only documentation. In *narrow* domains such as banking, it may be practical to write a program that extracts specifications from the old code. Such a tool would help immensely in decrypting modules to be rewritten.

8.3 Application Analyzer

Current code analyzers are adept at determining statistics for a given application (e.g., number of lines of code, call graph). Although this information is valuable, next generation tools could do much more. A code analyzer could inspect the modules in a large system and

determine metrics for difficulty of module migration. Such metrics are probably not statistical in nature, as are the current quality metrics used in software engineering. Rather, they are more akin to an expert system that knows about the difficulties in porting software to a new environment. Such a tool could assist the CMS and TPS migrations immeasurably.

8.4 Dependency Analyzer

It would be most valuable to have a tool that would inspect two modules, A and B, to ascertain if one did, or did not, depend on the other. Although it is straightforward to construct a call graph of a complex application, a powerful dependency analyzer would also inspect global variables and the reading and writing of auxiliary data structures, such as files. The best dependency analyzer would be able to guarantee that a module A was not dependent on another module B, so that the latter one could be replaced by a rewritten one without fear of application collapse.

8.5 Migration Schema Design and Development Tools

For both CMS and TPS, the structure and design of the legacy database had to be deduced largely from the legacy code. Due to legacy design techniques, data descriptions are distributed throughout the code and are often indistinguishable from application code. These factors and the lack of documentation make database design difficult. An additional database design problem is the design of the migration schema. The migration schema must provide a mapping from the legacy database to the target database. It could be considered to be a schema that integrates the legacy and target schemas. It would be a great help to have tools that assist with analyzing data definitions in legacy ISs and designing the migration schema. This may draw on results from schema integration and conceptual modelling research and products. Although the motivating problems are critical and some of the results useful, most results and related products are almost completely inadequate in providing solutions to real design problems such as those posed by real legacy ISs [BROD92]. For example, the most that existing products provide for automatic mapping between two schemas are suggestions that a data element in one schema matches an element in another schema based on simple text matching of the data element names.

8.6 Database Extractor

Legacy databases tend to be ill-structured and contain significant problems (e.g., TPS's broken pointer chains). They also tend to be vast. In TPS, a considerable amount of data was solely an artifact of the implementation and need not be migrated. It would be very effective to have tools that could extract data from legacy database, repair it if necessary, validate it against the migration database, translate it to the required formats, and load it into the target DBMS. Since legacy databases contain large amounts of corrupted data, such a tool should have powerful, automated mechanisms for dealing with exceptions, errors, and other problems. A growing number of "data scrubbing" products[13] that support some of the above functions are just beginning to be offered.

8.7 Higher Level 4GL

Our plans included incrementally rewriting most of CMS and TPS in a 4GL. To make this as financially digestible as possible, the leverage relative to COBOL and FORTRAN must

---

[13] E.g., Apertus Technologies' Enterprise/Sync product.

be made as large as possible. Although the current figure of merit (20::1 improvement) makes rewriting practical, it would be more attractive with higher level tools. What such a *5GL* would consist of is an open research question.

8.8 Performance Tester

Sizing of new hardware has been a *back of the envelope* exercise in the case studies. Although it is easy to distribute data and processing over multiple *jelly bean* systems, it would be desirable to have a *performance analyzer* that could predict response time for the CMS and TPS transaction loads using a new schema on new hardware. Such sizing studies are time consuming and could be aided by a powerful tool. The old adage "load a large database early and benchmark your high volume interactions" should be replaced by advice that is cheaper and easier to follow.

8.9 Application Cut-over

The CMS and TPS migration require several, critical cut-overs. In a large terminal (or PC) network, it is unrealistic to move instantly from one IS to another. A graceful way of moving to a new environment a few users at a time would be very helpful. Research is required into methods and tools to support smooth, iterative cut-over from legacy to target IS components while the IS is under continuous operation.

8.10 Distributed IS Development and Migration Environment

Distributed client-server computing is very popular for which an increasingly large number of concepts, tools, and techniques are being provided. It would be highly desirable to have a consistent environment that supports the development of distributed ISs. Since future distributed ISs will include components of legacy ISs and must be continuously evolved, the environment should also support the migrations described in this paper.

The environment should support the integration of arbitrary tools. It should provide a collection of tools to support the design, development, testing, deployment, and management of databases, applications, and user interfaces that will be distributed over a target environment. Besides the tools described above, the environment might also include tools for optimal distribution of code and data, database optimization, and distributed transaction design.

8.11 Migration Planning and Management

The composite IS that exists during migration is more complex than its component legacy or target ISs. This complexity requires careful management due to the mission critical nature of the ISs and to cost of errors. The key migration challenges involve planning and managing the process. These include: selecting the increments to be migrated, interleaving the largely independent steps, and sequencing the steps to diminish risk of failure. These are currently entirely intuitive decisions. As the many different interfaces, applications, and database migrations proceed in parallel and are cut-over in stages with both legacy and target versions, there will be many versions to manage. Migration planning for TPS and CMS are very complex processes that must be managed, since they are both for mission critical, operational ISs. CMS and TPS are both so complex that no one person or small group of people understands the entire system. Due to their critical nature, some aspects of migration planning and management should be automated. We believe that the infrastructure for future ISs must support continuous migration (i.e., evolution). Hence, support for evolution (called migration here) planning, management, and development (e.g., Section 8.10) must become an integral part of those infrastructures.

8.12 Distributed Computing

As mentioned above, legacy ISs should be migrated into computing environments and architectures that will avoid future legacy IS problems. We believe that in the future, ISs will interact cooperatively and intelligently, more like humans in organizations interact to accomplish tasks than is the case with currently independent ISs which we call *intelligent and cooperative information systems* (ICIS) [BROD92]. Although this vision goes far beyond that described above, it illustrates the presence of continual change and aspects of next generation ISs towards which current (legacy) ISs might be required to migrate.

9. CONCLUSIONS AND EPILOGUE

Future IS technology should support continuous, iterative evolution. IS migration is not a process with a beginning, middle, and end. Continuous change is inevitable. Current requirements rapidly become outdated, and future requirements cannot be anticipated. The primary challenge facing IS and related support technology is the ability to accommodate change (e.g., in requirements, operation, content, function, interfaces, and engineering). The incremental and iterative evolution of IS, as well as all other software, is being considered in many related areas such as software systems evolution [HUFF91] specifications, prototyping [GOLD92], and object-oriented design and development.

IS evolution and migration must be considered to be a dominate factor in IS life cycles. When ISs cost hundreds of millions of dollars and are mission critical for business processes that have values that are orders of magnitude greater, it is sheer folly to do otherwise. An appropriately designed target IS architecture can facilitate future evolution and integration with other ISs. If the target IS in not appropriately designed, it will become a next generation legacy IS to be migrated at additional cost.

One of the greatest contributions of database technology [SILB91] is data independence, the goal of insulating any changes to the database or to the applications from each other (i.e., ability to change one without affecting the other). We must now extend data independence to other aspects of ISs. The migration methods presented in this paper have addressed applications and interfaces in addition to databases. The goals in these areas, which correspond to data independence, could be called application independence, user interface independence, and IS interface independence. Research is required into core technologies (e.g., DBMSs, operating systems, programming languages, object-orientation, design and development methods and environments) to achieve these goals. Research is also required into systems architectures that will insulate all aspects of the systems from each other so that they can be modified independently to meet ever changing requirements and to take advantage of new technologies. This goal underlies application architectures (e.g., modularity, object-orientation) and next generation computing architectures, such as middleware [BROD93].

In this paper, we are attempting to contribute to the support of continuous evolution. We have proposed the Chicken Little strategy and a spectrum of supporting methods with which to migrate legacy ISs into advanced target environments. We illustrated the methods by means of two migration case studies of multi-million dollar, mission critical legacy ISs. Both migrations are following the proposed plans but are progressing at a very slow rate. After approximately one year, both migrations are still in the detailed planning phase of the first migration. This illustrates the extent to which such significant changes concerning mission critical ISs are only partly technical. Being responsible for the mission critical books of the bank or the books of the telephone company makes management very cautious. The reality is that the migration must be essentially risk free, effective in achieving its goals, and pay back in the short term. There are also psychological and skill barriers when moving from a well understood to a less understood environment.

The contribution of this paper is a highly flexible set of migration methods that is tailorable to most legacy ISs and business contexts. Each method consists of five basic steps:

- Iteratively migrate the computing environment.
- Iteratively migrate the legacy applications.

- Iteratively migrate the legacy data.
- Iteratively migrate the user and system interfaces.
- Iteratively cut-over from the legacy to the target components.

The critical success factor, and challenge in deployment, is to identify appropriate portions of the IS and the associated planning and management to achieve an incremental migration that is feasible with respect to the technical and business requirements.

A Chicken Little migration of a large legacy IS will take years. Before the migration is complete, unanticipated requirements will arise for further migration and evolution. Hence, the goal of the proposed methods and technologies is to support continuous, iterative evolution. We believe that the proposed gateways, whose primary function is transparent interoperability, will become the key elements in IS architectures that will support continuous evolution.

The methods and case studies presented in this paper concern incremental rewrites and not incremental re-engineering. However, it is often optimal not to rewrite portions of legacy ISs, sometimes impossible. Rather they should be integrated, transparently into the target distributed IS. Again, gateways are the primary means of such integration via interoperability (as described in [BROD93, BROD92, MANO92]) and treating them as *attached* systems.

Object-orientation was not explicitly addressed in this paper, nor was it considered in detail in the CMS and TPS migration plans. However, we do not preclude object-orientation. Rather, it is a real alternative to conventional languages and designs. The migration and target IS architectures are consistent with object-orientation due to the principles underlying their design. First, they are distributed, client-server architectures. Second, all components are intended to be separable (e.g., encapsulated) modules that are defined and used by other components in terms of interface functions or methods. Third, the gateways and architecture can and should (regardless of the use of object-orientation) facilitate communication between components by some form of message passing. With these principles and an appropriate choice of technologies, it may not matter what the internal design or implementation may be; hence object-orientation could be used as well. If object-orientation is not used initially, the migration architecture and the above principles are appropriate steps toward future conversion to object-orientation and a distributed computing architecture.

ACKNOWLEDGEMENTS

The authors acknowledge contributions to this work from a number of people, including Blayne Maring, George Kocur, Laurette Bradley, and Mark Hornick; and the following participants in the CMS migration study: Rich Delia, Jaime Carbonell, Carnegie Mellon University; Paula Hawthorn, Miro Systems, Inc.; Lawrence Markosian, Reasoning Systems; Sal Stolfo, Columbia University; Robert Wilensky, University of California; and John Davis, Anderson Consulting.

REFERENCES

[BREI90] Breibart, Y. et al., "Reliable Transaction Management in a Multi-database System," *Proc. 1990 ACM SIGMOD Conference on Management of Data*, Atlantic City, N.J., May 1990.

[BROD92] Brodie, M.L. and S. Ceri, "On Intelligent and Cooperative Information Systems,", *International Journal of Intelligent and Cooperative Information Systems* 1, 3 Fall 1992.

[BROD93] Brodie, M.L., "The Promise of Distributed Computing and the Challenge of Legacy Inforamtion Systems, in Hsiao, D., E.J. Neuhold, and R. Sacks-Davis (eds.), *Proc. IFIP TC2/WG2.6 Conference on Semantics of Interoperable Database Systems*, Lorne, Australia, November 1992, Elsevier North Holland, Amsterdam 1993.

[BROO75] Brooks, F., *The Mythical Man Month*, Addison-Wesley Publishing Co., Philippines, 1975.

[DATE87] Date, C., *Selected Readings in Database Systems*, Addison Wesley, Reading, Mass., 1987.

[ELMA92] Elmagarmid, A.K. (ed.), *Database Transaction Models For Advanced Applications*, Morgan Kaufmann, San Mateo, CA, March 1992.

[GARC87] Garcia-Molina, H., and Salem, K., "Sagas", *Proc. 1987 ACM SIGMOD Conference on Management of Data*, San Francisco, June 1987.

[GOLD92] Goldman, N. and K. Narayanaswamy, "Software Evolution through Iterative Prototyping," in *Proceedings of the 14th International Conference on Software Engineering*, Melbourne, Australia, May 1992.

[HUFF91] Huff, K.E. and O.G. Selfridge, "Evolution in Future Intelligent Information Systems," *Proceedings of the International Workshop on the Development of Intelligent Information Systems,* Niagara-on-the-Lake, April 1991.

[MANO92] Manola, F., S. Heiler, D. Georgakopoulos, M. Hornick, and M.L. Brodie, "Distributed Object Management," *International Journal of Intelligent and Cooperative Information Systems, 1,* 1, April 1992.

[OMG91a] Object Management Group, "The Common Object Request Broker: Architecture and Specification", OMG Document Number 91.12.1, Revision 1.1, December 1991.

[OMG91b] Object Management Group Object Model Task Force, "The OMG Object Model," draft 0.9, OMG Document Number 91.9.1, September 3, 1991.

[OZSU91] Özsu, M.T. and P Valduriez, *Principles of Distributed Database Systems*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[SILB91] Silberschatz, A., M. Stonebraker, and J. Ullman (Eds.) "Database Systems: Achievements and Opportunities", Communications *of the ACM,* Vol. 34, No. 10, October 1991.

[SKEE82] Skeen, D., "Non-Blocking Commit Protocols", *Proc. 1982 ACM SIGMOD Conference on Management of Data*, Orlando, Fla., June 1982.

[WACH92] Wachter, H., and Reuter, A., "The Contract Model", in *Transaction Models for Advanced Database Application*, Morgan-Kaufmann, 1982.