Repository Support for Metadata-based Legacy Migration

Sandra Heiler Wang-Chien Lee Gail Mitchell
GTE Laboratories Incorporated
40 Sylvan Road
Waltham, MA 02451
{sheiler,wlee,gmitchell}@gte.com

Abstract

Migrating legacy systems involves replacing (either wholly or partially) existing systems and databases, and complex transformations between old and new data, processes and systems. Correctly performing these activities depends on descriptions of data, and other aspects of the legacy and new systems, and the relationships between them, i.e., metadata. Metadata repositories provide tools for capturing, transforming, storing, and manipulating metadata. They can also store information for managing the migration process itself and for (re)use in other migrations. This paper discusses some of the issues that arise when migrating legacy systems and examines how repository technology can be used to address these issues.

1 Introduction

Modifications to software systems pose a constant challenge to businesses. The complexity of these modifications can range from routine software maintenance (to fix errors, improve performance, or to add or change functionality) to redefinition or replacement of entire business processes and, thus, of the software systems implementing them. Replacing (either wholly or partially) an existing software system is a *legacy migration*: the extant software systems are the legacy; the transformation of data and procedures from the old to new system is a migration process.

Two characteristics of legacy system migration create complexities that do not typically affect other software development:

- The legacy system provides an existing implementation of the target business process. Unlike new development, where the goal is to provide a working implementation at the end, a migration must maintain a working implementation at each step of the way.
- Legacy systems and databases have established relationships with other systems/databases of the enterprise. Data may be input to or output from other systems, or code may be reused in other applications. These relationships must be preserved or migrated during a migration effort.

For example, suppose an organization is migrating its financial systems to a new Enterprise Resource Planning (ERP) implementation (for example, [1]), first replacing the general ledger programs, then replacing the tax

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

accounting system, finally replacing the payroll system. The new general ledger programs might use data produced by the legacy accounts payable programs, and produce data that will be the new source of input for an executive reporting system. Similarly, the legacy HR system and new payroll system might use the same calculations for computing employee contributions to health insurance. At each stage the financial functions in the new system and the remaining functions in the legacy systems must operate properly and interoperate with each other. All the data used and produced by the financial systems must remain correct and consistent regardless of the migration stage and, of course, the systems processes must be correct and consistent at all stages.

Legacy system migration involves not just replacement of existing systems and databases, but complex mappings between old and new data, processes and systems. In most cases, there are no complete replacements during migration and, even once fully migrated, new business process will be implemented with a mix of old and new data and program code. As a result, legacy migration activities depend on descriptions of data, and other aspects of the legacy and new systems, and the relationships between them. This metadata allows designers and engineers to better understand what the systems do, how they do it, and when they do it, as well as what data the systems use or produce and where it comes from [2].

Supporting a legacy migration requires information about data and processes in the old and new systems. In addition, it requires information needed to move from one to the other and to integrate the remaining elements of the old with the elements in the new at each stage of the migration. This information can be represented as metadata describing, for example, components (e.g., usage, functions, interface), data (e.g., source, format, units), code (e.g., source language, change history, versions), processes (e.g., implementations, workflows, requirements), jobs (e.g., control language, performance statistics, hardware requirements), and relationships among the described elements (e.g., instance_of, part_of, computed_by, depends_on_completion_of).

Metadata repositories are designed specifically to store and manipulate these kinds of metadata. Repository technology provides tools for capturing metadata by extracting it from existing systems and transforming it for management in the repository. A repository can also store information about the migration process itself that can be (re)used in other migrations or maintenance, development,etc. projects.

In this paper we examine how repository technology can help with data transformation in the context of legacy system migration. Although we concentrate here on the transformation of data, many of the ideas will also apply to "transforming" procedures and processes. In the next section we outline the capabilities of available repository products. In Section 3 we present some of the issues related to data transformation when migrating legacy systems, and discuss how repository technology could be put in practice to facilitate migration. We conclude in Section 4, with a discussion of some additional advantages of using a repository during migration.

2 Repository Technology

A metadata repository consists of a set of software tools used for storage, search, retrieval, use and management of metadata. The components of a repository system include a metadatabase and metadata management system, an extensible metamodel and modeling tools, and tools for populating the metadatabase and integrating and accessing the metadata.

The metadatabase and management system. The repository includes a database of metadata and a DBMS to manage it. The repository DBMS provides standard database management facilities, such as persistent storage, data models, keys, certain types of queries, constraints and rules, transactions, data integrity and views, and also provides additional features for dealing with the specific requirements of metadata. Metadata-specific features include built-in complex relationships, navigation, support for long-running transactions, versioning and configuration management. Additional tools may be provided for defining and enforcing metadata security restrictions.

The repository DBMS is a logical system that may be physically instantiated as a single or distributed database. The repository database might be proprietary to a particular repository vendor, or implemented on top of standard database products. Typically a proprietary database will be object-based to support the extensive navigation required to implement and track metadata relationships and versions. A non-proprietary repository might be tightly integrated with a particular database product, or might work with a variety of products via standard interfaces. Requirements for the repository DBMS differ from those for an ordinary DBMS; in particular, the size of a metadatabase is typically orders of magnitude less than what must be supported by a regular DBMS.

Extensible metamodel and modeling tools. A repository system provides a meta-metamodel describing the things the repository needs to know about metadata (e.g., item types, attributes, relationships), and tools for building metamodels to include application-specific metadata descriptions. Generally, a repository product will also provide metamodels describing particular kind of metadata, for example, metamodels for different database schemas, software architecture specifications, or programming languages. Each metamodel provides built-in relationships specific to that type of metadata (e.g., a db2table *has_column* db2column). Metamodels can be modified to adapt to changes in the set of tools or applications supported by the repository, or to the specific characteristics of those tools. A repository product usually provides modeling languages and other tools to facilitate creation and modification of these metamodels.

Tools for populating the metadatabase. The metadata stored in a repository will come from a variety of sources, so a repository provides tools for extracting metadata from these sources (e.g., schemas from databases or data structure specifications from program code). Many software lifecycle management tools automatically store their output in a repository so that metadata generated in one stage of the cycle can be accessed or augmented in other stages of the cycle. For example, most CASE tools use a repository as a central location for capturing the connections between requirements, design and code. Other types of metadata sources do not provide an explicit representation of the metadata, so tools may be provided to extract information from each type of source and generate metadata for the repository[4]. For example, descriptions of a legacy file system might be extracted from COBOL code. Static tools (scanners or parsers) extract metadata from databases, ! ! program code or other sources; dynamic tools (buses) interact with CASE and other software life-cycle support tools to populate the repository as part of the process of using the tool. A specific scanner or bus is needed to obtain metadata from each different type of metadata source.

Tools for integrating metadata. Repositories support sharing metadata among different components or tools by providing the ability to define global models and integrated views of metadata from heterogeneous systems. Views can be defined to allow access to any item in a metadata repository by any application or user (subject to security restrictions) regardless of the originating source's platform, language, application system, etc. For example, a repository could provide a single view of metadata from existing DB2 files on a mainframe database, from old COBOL copybooks, and from a newly designed data model based on a CASE tool run on a UNIX system using C++. In addition, model definitions can be used to specify associations among repository elements that indicate semantic similarities among elements from heterogeneous sources, e.g., that a user-id in an ERWIN-generated data model and a customer_no in an Oracle database schema refer to the same data entities.

Tools for accessing metadata. A repository product also includes tools to facilitate easy access to metadata and customization of the repository. These typically include a user interface (GUI), (often) a Web interface for accessing metadata or Web tools for building such an interface, and APIs and/or a programming language for accessing metadata management functions and building additional, application-specific functionality. In addition, access applications such as impact analysis use the navigational facilities of the repository to provide information about the data relationships. For example, impact analysis tools can be used to estimate the scope of changes in a software maintenance project and determine which modules will be affected by proposed changes.

3 Issues in Data Transformation for Legacy System Migration

Transforming data as part of legacy system migration raises issues that are not necessarily of concern when data transformations are performed as part of other software engineering efforts. We focus here on three issues that typically affect transformation in (or are exacerbated by) migration efforts: data rationalization, interoperability, and process management. We discuss how these issues can be addressed (at least in part) using the tools of metadata repository technology.

Data rationalization. Unlike new software development efforts, legacy system migration starts with data that are part of the legacy systems delivering the specified functionality. An important aspect of migration planning is identifying appropriate sources of needed data and determining what transformations will be required to 1) be compatible with the new implementation, or 2) provide expanded functionality in the new system, and 3) support continuing relationships with other parts of the legacy.

For example, suppose the payroll portion of the migration example of Section 1 moves from a variety of different local systems to a centralized system. This requires identifying appropriate sources for the employee roster, time reporting, deductions, tax data, and so on. These sources may provide initial data for loading the new system, or may be used with wrappers as ongoing sources for the new system. In either case, the rationalization process must determine what kinds of transformations are necessary to align both the syntax and semantics of the legacy data items with the new requirements. Also, if multiple sources are available in the legacy for any part of the needed data, the data rationalization effort must determine which source or sources to transform.

Data rationalization can be supported by an inventory of source data elements, including data formats and meanings, the uses of the data, and other relationships. Such an inventory could be built by loading the repository[1] using data extraction tools that work with the input source types and the repository. It is important to note that the semantics of the data elements is intertwined with the functionality of the software components that create or use the elements. Thus, metadata describing the software components that process the data elements is needed to complete the picture of what data sources are available to support data rationalization.

The *process* of rationalizing data could also be supported using the repository. In particular, the metadata generated during the rationalization process – what decisions were made, why they were made, what items have been mapped, by whom, etc. – could be stored in the repository. This metadata could then be analyzed to determine discrepancies, for example, or referenced to assist in making further decisions about the data items or related items.

Interoperability. In any migration, the new system may receive data feeds from the legacy, may become the source of data for systems that are still part of the legacy, or may be required to trigger updates in the legacy. For example, the new payroll system may get its data feeds from the old time reporting system; the HR system (which is not part of the migration) may depend on the new payroll system to update employee benefits information such as vacation time. Since migration is an incremental process; the new implementation must exchange data with elements of the legacy in each delivered increment of the migrating system (including even the final product, because migration generally preserves parts of the legacy). Thus, support for interoperabililty requires detailed information describing the data elements of the legacy as well as the new system.

These data descriptions require both syntactic information (table layouts, units, encoding schemes, etc.) and semantic information (what an element ”means”, where it comes from, how it’s produced, how it’s used, etc.)[3]. For example, migrating payroll functions would require descriptions not only of the formats of employee records in the old HR system, but also descriptions of what constitutes a ”full-time” employee for benefits calculations, (e.g., appearance on the employee roster? having a particular form of employee number? reporting forty hours of time worked? Are different definitions of ”full-time” employee used for different purposes?)

Similarly, as parts of the functionality are migrated, new and old code modules must interoperate. For example, old modules for computing employee paychecks may need to interoperate with new check-cutting modules. Later in the migration, the old paycheck computational modules may be replaced and all will have to continue to interoperate with the old time reporting system. As another example, suppose the migration of the payroll system proceeds by geographic region. As each region is converted, payroll data from the migrated regions must be combined with similar data from any remaining regions for computations done in the (new) tax withholding modules. Support for interoperability requires metadata describing the components of the legacy and the migrated systems.

[1] Assuming, of course, that the system information was not loaded into the repository during earlier development or maintenance.

The repository can store descriptions of interfaces and relationships among the components of the old and new systems (e.g., between data elements and programs, or among software components that invoke one another) as well as descriptions of the semantics of the old and new data (and code) in the various migration increments. The repository also supports management of metadata describing software configurations, data flows, error conditions, and job structures and schedules. This could be used, for example, to determine when needed data has been updated or to obtain the transaction management requirements of the interoperating components.

Process management. Management of the migration process can be supported by metadata repository tools that track and control the configurations that form each of the increments and the changes to components from one increment to the next. Each release in a migration must be configured in such a way that necessary relationships between migrated and legacy system functionality and databases are preserved through the various versions of the migrated system, and the retirement of legacy components. For example, suppose that part of the migration process replaces project codes with a new coding scheme and that, in at least one system release, the project accounting modules use the old code and the time reporting modules use the new codes. This release cannot use a common project validation scheme, and historical records will include different code types. Managing the migration will require recording when the code changes occur, and mappings of modules to validation schemes in each version of a release. Testing procedures and error-correction mechanisms will also need to be aware of the various configurations. In addition, reporting systems will need to know the coding scheme changes to properly annotate reports.

Resolving process management issues depends on having information that describes the data and software components, and the business processes they implement. The information will change as the migration process unfolds, so resolving the issues requires not only up-to-date descriptions for each release version, but information about the process of change-when changes occur, the nature of the changes and the testing those changes trigger. In effect, what are needed are descriptions of the elements and the relationships among the elements, as well as descriptions of the changes in the elements and the relationships among the changes. Repositories generally provide versioning and configuration management tools that can document the contents of releases at each stage, thus maintaining an historical record of the migration. Moreover, they provide impact analysis tools that can then determine what elements were affected by each change and predict what will be affected by proposed ! ! changes.

4 Conclusions

Metadata repositories have long been used by CASE tools to store and manage descriptions of system components, and by data administrators to document information stores. More recently, they are being used to support integration of various tools, databases and applications, and their use is expanding to managing metadata for many more applications. We have shown how repositories can also be used to facilitate legacy migration by managing information about the existing and target systems and by providing tools to obtain, store, and analyze this information. As in new development efforts, design information in the repository can be used to generate some code or for decision-making in later stages of a migration. In addition, this information may be used to generate test cases and the repository can be used to track the testing process and associated scripts.

Metadata repository technology also helps to address specific issues of data transformation in the context of legacy migration:

1. Identifying appropriate sources of needed data and what transformations will be required to use that data in the migrated system is supported by data extraction tools and tools to assist with analyzing the metadata. The repository also provides a location for recording the relationships and transformations determined by a designer.

2. Interoperability among remaining legacy data (and systems) and migrated data is supported by a shared

model of data represented in the repository. In addition, descriptions of interfaces, component relationships, invocations, etc. can be stored in the repository, analyzed and used in designing and executing the integration.

3. Management of the migration process itself can be supported by metadata repository tools that track and control both the configurations that form each of the increments and the changes to components from one increment to the next.

A major advantage of using repository technology for storing metadata from a migration is the availability and usability of that information subsequent to the migration. For example, the information is useful in determining when elements of the legacy can be retired. Repository impact analysis tools can trace from new modules and data back to the legacy systems to see whether particular functions have been completely replaced (or replaced sufficiently) by the new systems. In addition, there is a continuing need for data from the migrated legacy to be combined with new data. The information gathered in the repository about the various systems and the migration itself can be used to compile accurate historical reports. Also, information gathered during the migration can be used in further maintenance of the system (correcting errors, new releases, adding enhancements, etc.) and may be useful in planning other migration projects.

Current applications of repository technology in software engineering typically address a single lifecycle stage or sequence of stages. Applications of repository technology to legacy migration tend to focus more broadly because they must include both the legacy and new systems and the components of each change as the migration process proceeds. However, they too tend to address only lifecycle stages directly related to the migration. Both applications would benefit by using the repository to obtain information about the existing environment and to store information during the project. We believe that significantly more benefits will accrue to the enterprise when metadata is shared across all stages of the lifecycle and across all projects. Our current work aims to engineer such complete information asset management using a repository.

References

[1] ASAP World Consultancy, Using SAP R/3. Que Corporation, 1996.

[2] S. Heiler, "Semantic Interoperability," ACM Computing Surveys, 27(2), 1995.

[3] V. Ventrone and S. Heiler, "Semantic Heterogeneity as a Result of Domain Evolution," SIGMOD Record, December, 1991; reprinted in Multidatabase Systems: An Advanced Solution for Global Information Sharing, A. Hurson, M. Bright, and S. Pakzad (eds), IEEE Computer Society Press, Los Alamitos, CA, 1993.

[4] J. Q. Ning, A. Engberts and W. Kozaczynski, "Automated Support for Legacy Code Understanding," in Communications of the ACM, 37(5), May 1994.