Carnegie Mellon
Software Engineering Institute

Pittsburgh, PA 15213-3890

Options Analysis for Reengineering (OAR): Issues and Conceptual Approach

John Bergey
Dennis Smith
Nelson Weiderman
Steven Woods

September 1999

Product Line Practice Initiative

Technical Note
CMU/SEI-99-TN-014

Contents

Abstract v

1. Introduction 1

2. Problem Definition 3

3. The Underlying Conceptual Model for Choosing a Reengineering Strategy 6

4. Reengineering Strategies and Options 11

5. Applying the OAR Method 16

6. Conclusion and Next Steps: Integrating Contextual Tradeoffs in OAR 17

References 20

List of Figures

Figure 1: “Horseshoe” Model for Integrating Reengineering and Software Architecture Views 7

Abstract

Organizations that own or use software assets require a structured and validated approach for making decisions on how to update, migrate, or reengineer their legacy assets. A model has recently been developed to understand technical transformations at different levels of abstraction. However, this model, which focuses on technical issues, is not yet accessible for decision-makers. This report outlines the foundation of a structured and coherent method, based on the “horseshoe” model, that will help practitioners make appropriate reengineering choices.

1. Introduction

All modern organizations own or use a considerable collection of software assets, and are thus in the software business. A major issue today is how to build and evolve software that can be managed as an investment that grows in value rather than a liability whose value depreciates over time.

Integration between systems has always been difficult; there is little systematic reuse of assets between systems; and new software quickly becomes a liability. To provide for greater reuse, a number of organizations have recently begun implementing software product lines that permit economies of scale and the capability to manage software as an enterprise investment.

The emerging focus on software architectures and product lines has led to an emphasis on the evolvability of software systems. Systems with a well-conceived architecture allow the software to interact across well-defined interfaces without regard for internal implementation details. However, most product lines start with legacy systems that need to be updated to enable them to interact as well-defined components. Although it is possible to update legacy systems through reverse engineering, these techniques are costly. They also usually focus at the code level, and as a result do not usually provide leverage for updating to modern architectures. However, recent architectural extraction techniques [Kazman 97] provide a basis for extracting the as-built architecture from the existing code, and using this information as a foundation for further evolution.

In conjunction with the architectural extraction work, Woods developed a conceptual "horseshoe" model that distinguishes different levels of reengineering analysis and provides a foundation for transformations at each level, especially for transformations to the architectural level [Woods 99]. This model describes the rich set of technical choices that reengineers make. However, because of its technical focus, it has not been accessible to decision makers in a form that can assist them in deciding on complex options regarding the future of their legacy systems.

We take the horseshoe model as a starting point and propose a structured and accessible vehicle for making informed choices in a variety of real world situations. This method, "Options Analysis for Reengineering" (OAR) is a reengineering decision aid, grounded in the technical underpinnings of the horseshoe model. It provides a structured and coherent method for making technical, organizational, mission, and programmatic choices in practical reengineering decision making.

This paper introduces the need for OAR, its conceptual foundation, and the problems it addresses. In the future the OAR approach will be refined and codified through work with organizations that implement and use reengineering techniques at different levels of abstraction. Feedback is actively solicited and will be factored into the development of the OAR approach.

Section 2 outlines the purpose of OAR and its relationship to current reengineering problems. Section 3 discusses the levels of reengineering analysis defined by the horseshoe model. Section 4 highlights the circumstances under which different levels of analysis can be applied. Section 5 introduces some of the potential applications of the OAR approach. Section 6 outlines the issues involved in integrating the programmatic, organizational, and technical facets of reengineering decision making that will be the focus of OAR.

2. Problem Definition

Software reengineering transforms an existing design of a software system (or elements of that system) to a new design while preserving the system's intrinsic functionality. The purpose of OAR is to provide a structured and coherent method to help practitioners make appropriate reengineering technology choices.

2.1 What is OAR Trying to Accomplish?

The OAR method will enable a reengineering practitioner to answer the following types of questions:

- What reengineering options/approaches are applicable to my problem?
- What do each of them entail?
- Where and how do architecture considerations come into play?
- What advantages (or disadvantages) does one option/approach have over another?
- How are these options/approaches interrelated?
- What are the organizational, programmatic, and technical implications of each?

OAR presents a "big picture" view of *reengineering software analysis and transformation options*. OAR synthesizes reengineering approaches across several levels of system understanding (e.g., source code, function-level, and architectural). It details the type of design/implementation options that can guide the selection of a reengineering strategy. This enables a practitioner to develop a coherent approach and gain a better understanding of the tradeoffs that are involved as well as organizational and programmatic issues and their implications.

OAR's underlying model

- combines reengineering and architectural views of software analysis and evolution with defined mappings between these views
- classifies and stratifies reengineering analysis and implementation options/approaches into distinct layers with explicit mapping between layers
- provides key information for making informed choices about the appropriate time and circumstances in which to use each of the reengineering options/approaches

- codifies technical and non-technical issues and risks for each of the reengineering options/approaches
- relates organizational and programmatic factors to *reengineering option* decision making in a system and product line context

By combining these features, the OAR method provides a unified approach for analyzing reengineering options and evaluating candidate reengineering strategies.

2.2 Why Is OAR Needed?

Reengineering is complex. By carefully analyzing candidate reengineering options and strategies that affect the interests of many stakeholders, it requires making non-trivial tradeoffs about technical, programmatic, and organizational considerations.

Reengineering decision-making requires

- considering a diversity of (reengineering) options, each of which may involve significant tradeoffs
- performing analyses and interacting with experts to compensate for a lack of up-to-date legacy system requirements/design documentation
- resolving the uncertainties about the implementation of a legacy system including its functionality, integrity, and quality attributes
- obtaining extensive quantitative and qualitative data on which to base decisions
- exploring the impact of reengineering from the perspective of multiple stakeholders and resolving conflicts stemming from a fracturing of corporate knowledge and expertise
- accommodating organizational and programmatic constraints and coalescing decision-making across the organization

Reengineering can be especially challenging because it often attempts to evolve a legacy system when the system itself has reached a point where maintenance and enhancement practices are no longer adequate. The technical challenges require an effective method for performing the following tasks:

- analyzing critical requirements/design information (from documentation and interaction with system experts) to derive or reconstruct an “as-designed” baseline
- reverse engineering the legacy system to derive a robust representation of the “as-built” system baseline.
- identifying and resolving discontinuities or incompatibilities between the as-designed system representation and the as-built system baseline
- analyzing reengineering options to select a reengineering strategy that defines an effective mix of system transformations for implementing the “as desired” system

The adoption of a semi-formal method or approach for reengineering is critical for any organization, at a minimum. The method should be applicable to reengineering a system at different levels of design abstraction. It should accommodate the full range of reengineering activities including

- gaining a comprehensive system understanding (reflecting multiple design/implementation views)
- exploring candidate options
- making tradeoffs leading to the selection of a viable reengineering strategy

2.3 How are Reengineering Decisions Made Without a Semi-Formal Method?

In the absence of a semi-formal reengineering method, organizations often apply “seat-of-the-pants” decision-making in response to schedule pressure and other programmatic and organizational concerns. Not surprisingly, the most commonly cited reason for reengineering is to save time and money. Without an established method, organizations often choose the cheapest near term alternative. However, when the project is in the implementation stage, these decisions often prove to be shortsighted when the project misses deadlines and has cost overruns.

Bergey cites *the top ten reasons for the failure of reengineering projects* [Bergey 99]. Of these top ten reasons, the following are relevant to the issues that OAR addresses:

- The organization inadvertently adopts a flawed or incomplete reengineering strategy.
- The organization does not have its legacy system under control.
- There is inadequate planning or inadequate resolve to follow the plans.
- There is no notion of a separate and distinct reengineering process.
- Software architecture is not a primary reengineering consideration.
- There is too little elicitation and validation of requirements.
- Management predetermines technical decisions.

For successful reengineering the articulation of options and tradeoffs for making decisions is critical. There needs to be a systematic codification of the technical and organizational issues that are relevant to making informed choices.

Our experience in collaborating with customers on reengineering projects has shown that the lack of a semi-formal reengineering method results in the failure to identify risks and thus can potentially derail migration efforts.

3. The Underlying Conceptual Model for Choosing a Reengineering Strategy

The reengineering process involves the disciplined evolution of an existing legacy system to a new “improved” system. In its most fundamental form this process includes

1. analysis of an existing system, resulting in one or more logical descriptions of the system
2. transformation of those logical descriptions into new, improved logical descriptions
3. development of the new system based on these new logical descriptions

The horseshoe model (Figure 1) combines reengineering and architectural views of software analysis and evolution. Its relationship to organizational and management issues is described in Sections 4 and 5.

3.1 The Horseshoe Model

The three basic reengineering processess—i.e., analysis of an existing system, logical transformation, and development of a new systems—form the basis of the horseshoe. The first process goes up the left leg of the horseshoe, the second goes across the top of the horseshoe, and the third goes down the right leg of the horseshoe. The richness of the horseshoe comes in the three levels of abstraction that can be adopted for the logical descriptions. Conceptually, it can be thought of as a set of nested horseshoes. The logical descriptions can be artifacts as concrete and simple as the source code of the system or as abstract and complex as the system architecture. The purpose of the visual metaphor is to integrate the code-level and the architectural reengineering views of the world.

The three basic reengineering processes are briefly outlined below as they fit into the horseshoe model.

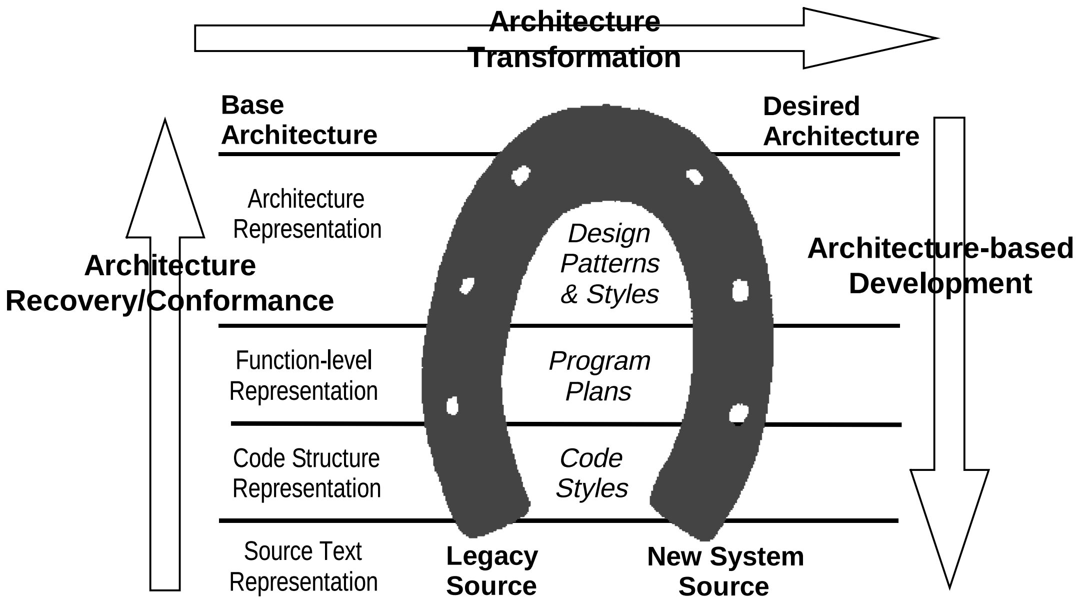
Architecture Transformation
Base Architecture
Desired Architecture
Architecture Representation
Design Patterns & Styles
Architecture Recovery/Conformance
Architecture-based Development
Function-level Representation
Program Plans
Code Structure Representation
Code Styles
Source Text Representation
Legacy Source
New System Source

Figure 1: Horseshoe Model for Integrating Reengineering and Software Architecture Views

In its purest and most complete form (represented by the large outlined arrows), the first process recovers the architecture by extracting artifacts from source code. This recovered architecture is analyzed to determine whether it conforms to the as-designed architecture. The discovered architecture is also evaluated with respect to a number of quality attributes such as performance, modifiability, security, or reliability.

The second process is architectural transformation. In this case, the as-built architecture is recovered and then reengineered to become a desirable new architecture. It is re-evaluated against the system's quality goals and subject to other organizational and economic constraints.

The third process of the horseshoe uses Architecture-Based Development (ABD) [Bass 99] to instantiate the desired architecture. In this process, packaging issues are decided and interconnection strategies are chosen. Code-level artifacts from the legacy system are often wrapped or rewritten to fit into this new architecture.

As mentioned earlier, the trip around the outside of the horseshoe represents the most abstract and purest form of reengineering. In practice there are two additional shortcuts that cut across the horseshoe and that enable one to get from the as-built system to the as-desired system. These "shortcut" paths across the horseshoe can represent pragmatic choices based on organizational or technological constraints, such as reengineering tools available. In these cases our analysis process does not result in the system architecture representation, but in lower level artifacts that may be closer to the source code than the architecture.

3.2 Three Levels of the Horseshoe

There are three levels of the horseshoe:

1. "Code-Structure Representation," which includes source code and artifacts such as abstract syntax trees (ASTs) and flow graphs obtained through parsing and rote analytical operations.
2. "Function-Level Representation," which describes the relationship between the programs functions (calls, for example), data (function and data relationships), and files (groupings of functions and data).
3. "Concept" level, which represents clusters of both function and code level artifacts that are assembled into subsystems of related architectural components or concepts.

The complete model has transformations not only at the architectural level, but also at two subsidiary levels as well. Next we will provide some simple examples of transformations at each level.

Source Level

At the source level there are actually two sub-levels, textual (or string-based) transformations and transformations based on the syntax tree. While some reengineering approaches distinguish between "1A" textual (syntactic) and "1B" AST-based (semantic) transformations, the horseshoe model considers them both in the context of code-structure. We will refer to these two sub-levels as level 1A and level 1B, respectively.

Textual Transformations

At level 1A the textual transformations are done through various simple string matching and replacement approaches. For example, many of the Year 2000 (Y2K) remediation efforts look for patterns in the source code that represent manipulation of two digit year representations (for example, 99 for 1999) and then replace them with code that manipulates four digit dates. Very simple tools that perform matching and replacement handle this type of task. Other examples include textual transformations of names or keywords when porting a system from one platform or operating system to another. Level 1A transformations are relatively simple, straightforward, and inexpensive relative to tool support or personnel support. They are chosen by organizations when the problem is sufficiently simple, or when a "quick and dirty" result is required.

Syntax Tree Transformations

At level 1B code-structure transformations based on syntax trees (AST-based transformations) support changes that are immune to surface syntax variations. Thus, the source code representation is independent of the expression syntax or the way that programming languages represent numbers. Code-structure transformations based on syntax trees are often used to do ports to a new implementation language (e.g., COBOL to C++) or to automatically make changes to the code (e.g., more sophisticated approaches to the Y2K problem).

Function Level Transformations

Function level transformations (level "2") have to do with repackaging of functionality (e.g., moving from a functional design to an object-oriented design or moving from a relational database model to an object database model). Wrapping, the encapsulation of a given module of functionality for a different contextual environment, is an example of a functional level transformation. Functional level transformations go beyond simple code-structure transformations, but do not go as far as architectural transformations. They are chosen when large units of functionality can be salvaged by placing them into a new context.

Architecture Level Transformations

Architectural level transformations (level "3") involve changing the basic building blocks of the architecture. These include the basic patterns of interaction including the types of components, the connectors used, the allocation of functionality, and the run-time patterns of control and data exchange. The architectural level is the most abstract and far reaching of the transformations. Architectural level transformations are made when major structural change is called for due to major modifications or major deficiencies in the legacy system. Level 3 transformations generally call for major commitments of time and resources, but also reap the greatest benefits as discussed in Section 4.

Interaction Between Levels

Code-structure transformations can take place without functional level changes or architectural changes. In addition, functional level changes can take place without architectural changes. The lower level transformations support the upper level transformations. With these multi-level views of transformations, architectural transformations are normally the context in which lower level transformations take place. However, upper level transformations do not support lower level transformations because the lower-level transformations can take place independently of upper-level transformations. In fact, one of the primary purposes for the horseshoe model is to raise the level of abstraction and bring architectural notions to the task of reengineering.

As an example of how these levels interact, consider the migration of a client/server architecture to a CORBA-based architecture. This is a change that cannot take place only at the code-structure level or even the functional level because the basic building blocks and connections between those building blocks are going to change. However, at the functional level there can be opportunistic wrapping strategies to use existing portions of the code in a new context. At the code structure level there can be changes to support different programming languages or different computing platforms.

3.3 Implications of the Horseshoe Levels for Reengineering

Significant system understanding, as well as architecture expertise, is required to move up levels of abstraction in the horseshoe. At level one, the system source code (or the syntax tree) is the primary source of information. The source code is usually available and the compilers used to translate the source code can produce syntax trees. Very little architectural information is required and very little architectural expertise is needed.

As we go to levels two and three, the information is more abstract. If it exists at all, it exists in system documentation or in the head of the architect. At levels two and three, the as-designed system may not match the as-built system. The extraction process involves interaction with experts and analysis of source code. The extracted "base architecture" is then mapped to the extracted source information. This often demonstrates the gaps which have evolved between the "expert" view of a system and its realization.

At the architecture level, OAR has two possible goals:

1. validation of implementation against design
2. migration from an old design to a new one

On the one hand, architectural analysis can reassure an organization that existing non-functional quality analyses are valid by recognizing conformance between implementation and described architecture. On the other hand, it can support the process of migrating from an old architecture to a new one that embodies new quality tradeoffs.

While Figure 1 describes the underlying horseshoe model, the OAR method will, at an abstract level, support the explicit mapping between layers. Existing reengineering techniques can now be seen as "transformations" across layers, sometimes bridging higher layers, and sometimes failing to do so.

OAR provides its greatest value when a new architecture is developed. In this case the new architecture is populated with the artifacts from the "as-recovered" legacy architecture, and a new system is instantiated using the techniques of Architecture-Based Development.

OAR will explicitly detail all the options for reengineering at each level of the horseshoe within the organizational context.

4. Reengineering Strategies and Options

In an ideal world, reengineering efforts would look quite different from the way they look in the real world. In the ideal world, organizations would have unlimited time, money, skills, and tools available and would have the flexibility to implement the maximum functionality with the best performance, reliability, availability, and security. OAR is only necessary in the real world where there are engineering, organizational, and financial constraints placed on the problem of moving from an existing system to a “better” one.

The application of reengineering technology to the task of architecture recovery and reconstruction is relatively new. There have not yet been efforts to address precisely when or how to use these techniques. In addition, the differential costs, risks, and rewards associated with architecture reconstruction or with more traditional approaches to reconstruction have not been assessed. As a result, many current efforts have taken place below the level of architecture reconstruction. In this section we will explore the range of choices that will be part of an OAR.

The articulation of tradeoffs for making decisions is critical, but there has not been any codification of the technical and organizational issues relevant for making informed choices. Several simple sets of options that correspond to the three levels in the horseshoe representation (starting from the bottom up) include

- simple source code transformations such as those widely used to address Y2K problems
- both “black box” (wrapping approaches) and “white box” redesign of functional code structure of individual components to accommodate new or modified requirements
- moving to an architectural level of understanding in order to develop core assets for a product line

The first alternative is relatively low-cost and requires the least knowledge about the legacy system and a relatively small amount of skill and organizational sophistication. These are the primary reasons that it is generally chosen for Y2K problems. The second alternative requires a greater understanding of the application (at least at the system interface level) and moderate skill and tool capabilities. The third alternative requires the greatest amount of knowledge about the architecture of the legacy system and requires highly developed skills and organizational sophistication. It is most desirable for organizations moving to product line strategies.

4.1 Overarching Considerations

If the goal is simply to replace a legacy system with one that has improved functionality or better performance, then modest means can often be employed with no ancillary benefits. If the goal is to start a product line then more deliberation is necessary. A product line is defined as a group of products sharing a common, managed set of features that satisfy the needs of a selected market or mission area. Product line practices incorporate techniques for finding and exploiting system commonalities and for controlling variability and then incorporating them as standard software engineering practices. The SEI has developed the Framework for Software Product Line Practice [Clements 99] to codify the software engineering, technical management, and organizational management techniques for successful product line practices. Inherent in this framework is the assumption that products do not start as "greenfield" development, but rather they evolve from legacy systems.

Specific to reengineering, Bergey describes a broad set of enterprise-wide management and technical issues [Bergey 97]. This work provides a characterization of the global environment in which systems evolution takes place and provides insights into the activities, practices, and work products that shape the disciplined evolution of legacy systems. It identifies important global issues early in the planning cycle and provides checklists for testing progress. The OAR will draw on this work, as well as the work of Woods [Woods 99], to formulate decision-making guidance for different types of circumstances.

The following sections discuss circumstances where different types of transformations at the three levels of the horseshoe are applied.

4.2 Code Structure Transformations

When are code structure transformations appropriate? They are appropriate when

- no structural changes at the architectural level are anticipated or desired
- the source code is the primary artifact and little or no architectural information is present
- cost and time are severely limiting constraints
- organizational and technical sophistication is limited
- the risks of structural change outweigh the benefits

What are some of the major issues for code structure transformation?

- the ability to use tools for manipulation of text and syntax trees
- availability of programmers
- being able to isolate the required changes to a specific set of components/code segments
- regression testing based on legacy system
- configuration control of source code

Code structure transformations are the “quick and dirty” form of software evolution. They are often associated more with maintenance activities than with reengineering activities. If the software is old and “crufty” and none of the original designers is around to explain the structure of the code, this may be the way to evolve the system. The Y2K problem is the perfect example. The problem has not gained attention because it is a particularly complex problem. Rather, it has gained attention because it is so pervasive. A whole industry has grown up around the simple problem of making fairly simple local changes to code that is, in many cases, decades old. In fact, because of the Y2K problem, some significant strides have been made in the program understanding tools available for reverse engineering at the lowest level of abstraction [Tilley 98].

It should be noted that with this limited approach, if you start with stovepipe software you retain stovepipe structures. If the software started out as brittle, it remains brittle. If it is unstructured and undocumented, it remains unstructured and undocumented. In short, there is very little rehabilitation that takes place for software that is transformed in this way. At most, there may be some additional low-level information about the components and a better sense of configuration control as a result.

4.3 Function-level Transformations

When are function-level transformations appropriate? They are appropriate when

- some structural changes are necessary
- stucture of the systems and their interfaces is well documented or can be extracted
- cost and time are moderately limiting constraints
- organizational and technical sophistication is moderate
- the risks of structural change are moderate

What are some of the major issues for function-level transformation?

- tools for recognizing modules and interfaces
- availability of system analysts and programmers
- mining of assets for reuse
- wrapping of mined assets

Functional-level transformations are a medium-level approach to code transformations that are often associated with changes in technology while maintaining basic functionality. Whether changing to a new database system, changing from a functional to object-oriented paradigm, changing from client-server to object request broker, or changing from off-line processing to electronic commerce, some structural change is required, primarily in the interfaces to subsystems. To make functional-level transformations, the structure of the system must be understood—at least at the interface level. It may be possible to “wrap” modules of functionality when the internals are not well understood for use in another context. Some of the issues associated with functional-level transformations are discussed in *Approaches to Legacy System Evolution* [Weiderman 97].

Functional-level transformations start to provide some major benefits. Software has been restructured for different purposes. The changes are more than cosmetic changes. In fact, the changes can be precursors to more substantive architectural changes. Software becomes easier to maintain. In comparison to code-level changes, the mined and rehabilitated assets become better structured, better documented, and provide more clearly defined and more general interfaces.

4.4 Architectural Transformations

When are architectural transformations appropriate? They are appropriate when

- product line aspirations are high
- system architecture is well documented or derivable with the help of a system architect
- cost and time are not severely limiting constraints
- organizational and technical sophistication is high
- the business case risk of continuing to maintain stovepipe systems is greater than the risk of developing product lines

What are some of the major issues for architectural transformations?

- tools for extracting architecture and recognizing quality factors
- availability of system architects, analysts, and programmers
- mining of assets for product line core assets
- architecture-based development

Many companies are finding that the only way that they can compete in a software-intensive business is to find solutions that they can reuse at a high level of abstraction. This is not component reuse, but reuse at the level of software architecture. Software architecture recovery is the first step toward developing a core set of assets that can be used in Architectural Based Development.

The SEI has several methods and tools that can assist in architecture recovery. The Architecture Tradeoff Analysis Method^{SM} (ATAM^{SM}) [Kazman 98] is a method that draws together all the stakeholders of a particular software system using scenarios to analyze the architecture with respect to multiple quality attributes. It is a people-oriented, rather than tool-oriented, approach. “Dali,” on the other hand, is a powerful, integrated tool set that works with source code as well as the system architect to extract the essence of the system architecture [Kazman 97].

More recently an approach has been developed to identify and catalog architectural styles using Attribute-Based Architectural Styles (ABASs)[1].

One aspect peculiar to the risk-reward structure of architectural transformation is that mined assets, just as newly developed assets, can be sub-optimal for specific tasks as compared to single system use. What is important for mined architectural assets, just as for newly developed assets, is that they are optimal for the product line over the long term, not optimal for one particular product. This optimality covers not only an asset's performance, but particularly its maintainability over time and its suitability over time for the entire product line.

4.5 Assessing the Alternatives of the Horseshoe

The horseshoe model is not a replacement for reengineering or architectural analytic techniques, but rather a description of how they are complementary in the process of controlled software evolution. It emphasizes that one size does not fit all situations and organizations. The leverage gained from the combination of powerful "low-level" reengineering tools and "high-level" analytic models can be seen from the combination of the architectural methods and tools described above. Analysis efforts demonstrate that new system quality tradeoffs can be "engineered-in" if one considers the non-functional qualities carefully in the architectural design of a system (or re-design of a legacy system). Bergey's work [Bergey 97] provides guidance on organizational and technological alternatives for those considering large-scale system reconstruction efforts.

The synergy between these views is demonstrated in the promise of emerging tool sets, such as Dali, that utilize traditional reengineering analysis in service of architectural reconstruction, representation and architecture-based development activities. Mining of assets at the architectural level has strong potential for gaining leverage from existing systems, as well as for forming potential core assets for a product line.

Only by assessing all the risks and rewards of the various techniques, methods, and tools can the proper engineering tradeoffs be assessed and evaluated. This will be the challenge of OAR.

[1] Klein, M. & Kazman, R. *Attribute-Based Architectural Styles* (CMU/SEI-99-TR-022). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University. Expected publication date: November, 1999.

5. Applying the OAR Method

OAR can be used for two different reengineering purposes:

1. to evaluate a proposed reengineering strategy
2. to explore candidate reengineering options

Both of these rely on analyzing reengineering options corresponding to the three levels of analysis, in the horseshoe representation (described in Section 3).

The horseshoe model provides a common frame of reference for organizing and obtaining the essential information needed to guide the decision making process. For each of the three levels of analysis, OAR will identify approaches that reflect proven practices applicable to the particular level. These options may include traditional as well as novel reengineering approaches extracted from successful case studies.

At the less abstract levels of analysis, options will be better defined, such as in the case of Y2K remediation for which solutions abound. These solution approaches reflect traditional reengineering techniques that are representative of the current state of reengineering practice. At the more abstract levels of analysis, options may not be as well codified because architecture reconstruction and architecture evaluation are leading edge technologies that are still evolving. In general the less abstract level levels tend to be more "tool oriented" while the higher levels are more "expertise-oriented."

There will be cases where the architecture may not be recoverable because of the lack of expert knowledge and pertinent documentation. This applies to the as-designed architecture and the as-built architecture of the legacy system. In such cases, the reengineering options at the architecture level will not be implementable by the practitioner and may preclude being able to effectively reengineer the system. From a product line standpoint, it may also preclude, or severely limit, the mining of the legacy system(s) to obtain start-up core assets. The importance of obtaining such insights—early in the reengineering processs—before full-scale implementation begins cannot be over emphasized.

6. Conclusion and Next Steps: Integrating Contextual Tradeoffs in OAR

In evaluating candidate options and selecting a reengineering strategy, there are not only technical tradeoffs, but tradeoffs involving organizational and programmatic considerations. OAR will codify organizational and programmatic issues associated with each of the options to guide the reengineeering decision making process. The codification of these issues can be expressed in terms of the prerequisites, constraints, risks and rewards, and other guidelines that apply to a specific option. OAR will outline these factors across the three levels of analysis. This will assist a practitioner in making informed choices about the appropriateness of using a specific option, or set of options.

The objective is to identify choices, illuminate benefits and risks, and provide insight into the implications of selecting a particular option. By considering these factors across the three levels, a practitioner can obtain a “big picture” view of a candidate reengineering approach. This information can provide the technical basis for

- determining which reengineering options are feasible and their potential impact
- establishing the advantages of one approach over another in a particular business context
- determining when it may be more advantageous to adopt an acquisition-based approach versus development-based approach
- developing a business case for reengineering

One of the challenges facing OAR is *to integrate these contextual factors in assisting a practitioner in resolving the overarching considerations leading to selecting a particular reengineering strategy.* The type of high-level organizational and programmatic issues that practitioners need to consider in their decision making include the following:

1. *Time and schedule considerations*

- How quickly do you need it?
- How much are you willing to give up to get it that quickly?
- Do the selected options allow for reengineering in discrete increments (each providing increasing capability) to “buy time” before the entire effort must be completed?

2. *Cost considerations*

- Is adequate funding available?
- Is there a cost allocation for each improvement?
- Can funding be tied to achieving pre-approved “system capability” milestones?
- What approach will yield the best value (e.g., Total projected savings/total cost)?
- What are the relative costs of continuing with the legacy system?

3. *Risk considerations*

- Are the risks associated with the overall approach acceptable?
- Can choosing another option or set of options substantially reduce risk?
- Can the risks be mitigated in a cost effective manner?
- In a worst case scenario, do any of the options involve potentially catastrophic risks?
- Can the acquisition of products or services reduce risk?

4. *Organizational sophistication*

- Is a suitable supporting infrastructure in place?
- Are defined and repeatable processes in place to support the selected options?
- Is there product line expertise?
- Is there architectural expertise?
- Is there reengineering tool expertise?
- Are there ample in-house resources?
- Acquisition expertise?

5. *Political climate*

- Is there a strategic plan and a realistic set of goals and objectives?
- Is there high-level support for long-term solutions?
- Are the resources for re-architecting viewed favorably?
- Are risks linked to program impact?
- Is the selected approach commensurate with stakeholder needs?

6. *Product line aspirations*

- What is the lifetime of the product?
- Is the domain well understood?
- Is there a need for multiple versions of similar products?

- Are the commonalities and variabilities well understood?
- Can funds be effectively pooled across participating projects?

7. *Contextual considerations*

- What is an appropriate juncture (e.g., the interface and iteration means) between OAR and those activities that typically fall under (reengineering) project planning?

In conclusion, we have taken a powerful technical model that characterizes reengineering approaches at different levels of analysis. Based on this model we have outlined a reengineering decision aid, "OAR," that uses technical, management, and organizational insights in providing a structured and coherent method for making practical reengineering choices. This technical note outlines the basic approach and issues of OAR. These concepts will evolve and be validated through ongoing work with organizations. The approach will be disseminated through technical notes and guides over the next several years. These types of issues and tradeoffs will be addressed as OAR evolves into a systematic decision-making tool and it becomes codified through experiences with a variety of organizations. During the next year outcomes of this work will include lessons learned from case studies that focus on transformation and extraction issues. Over the next two years, a fuller description of the model and a reference guide will be developed.

References

[Bass 99] Bass, L. & Kazman, R. *Architecture-Based Development* (CMU/SEI-99-TR-007, ADA366100). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1999. Available WWW: <URL: http://www.sei.cmu.edu/publications/documents/99.reports/99tr007/99tr007abstract.html>.

[Bergey 97] Bergey, J.; Northrop, L.; & Smith, D. *Enterprise Framework for the Disciplined Evolution of Legacy Systems* (CMU/SEI-97-TR-007). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1997. Available WWW: <URL: http://www.sei.cmu.edu/publications/documents/97.reports/97tr007/97tr007abstract.html>.

[Bergey 99] Bergey, J.; Smith, D.; Tilley, S.; Weiderman, N.; & Woods, S. *Why Reengineering Projects Fail* (CMU/SEI-99-TR-010). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1999. Available WWW: <URL: http://www.sei.cmu.edu/publications/documents/99.reports/99tr010/99tr010abstract.html>.

[Clements 99] Clements, P. et al. *A Framework for Software Product Line Practice – Version 2.0.* Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, July 1999. Available WWW: <URL: http://www.sei.cmu.edu/plp/framework.html>.

[Kazman 97] Kazman, R. & Carriere, S.J. *Playing Detective: Reconstructing Software Architecture from Available Evidence* (CMU/SEI-97-TR-010, ADA362725). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1997. Available WWW: <URL: http://www.sei.cmu.edu/publications/documents/97.reports/97tr010/97tr010abstract.html>.

[Kazman 98] Kazman, R.; Klein, M.; Barbacci, M.; Longstaff, T.; Lipson H.; & Carriere, S.J. *The Architecture Tradeoff Analysis Method* (CMU/ SEI-98- TR-008, ADA350761). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1998. Available WWW: <URL: http://www.sei.cmu.edu/publications/documents/98.reports/ 98tr008/98tr008abstract.html>.

[Kazman 99] Kazman, R.; Woods, S.; & Carriere, S.J. "Requirements for Integrating Software Architecture and Reengineering Modes: CORUM II," 154-163. *Proceedings of the 5th Working Conference on Reverse Engineering (WCRE-98)*. Honolulu, Hi., October 12-14, 1998. Los Alamitos, Ca.: IEEE Computer Society, 1998.

[Tilley 98] Tilley, S. *A Reverse-Engineering Environment Framework* (CMU/SEI-98-TR-005, ADA343688). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1998. Available WWW: <URL: http://www.sei.cmu.edu/publications/documents/98.reports/ 98tr005/98tr005abstract.html>.

[Weiderman 97] Weiderman, N.; Bergey, J.; Smith, D.; & Tilley, S. *Approaches to Legacy System Evolution* (CMU/SEI-97-TR-014). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1997. Available WWW: <URL: http://www.sei.cmu.edu/publications/documents/97.reports/ 97tr014/97tr014abstract.html

[Woods 99] Woods, S.; Carriere, S.J.; & Kazman, R. "A Semantic Foundation for Architectural Reengineering and Interchange," 391-398. *Proceedings of the International Conference on Software Maintenance (ICSM-99)*. Oxford, England, August 30-September 3, 1999. Los Alamitos, Ca.: IEEE Computer Society, 1999.

REPORT DOCUMENTATION PAGE			*Form Approved* *OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.			
1. AGENCY USE ONLY (LEAVE BLANK)		**2. REPORT DATE** September 1999	**3. REPORT TYPE AND DATES COVERED** Final
4. TITLE AND SUBTITLE Options Analysis for Reengineering (OAR): Issues and Conceptual Approach			**5. FUNDING NUMBERS** C — F19628-95-C-0003
6. AUTHOR(S) John Bergey, Dennis Smith, Nelson Weiderman, Steven Woods			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213			**8. PERFORMING ORGANIZATION REPORT NUMBER** CMU/SEI-99-TN-014
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) HQ ESC/DIB 5 Eglin Street Hanscom AFB, MA 01731-2116			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES			
12.A DISTRIBUTION/AVAILABILITY STATEMENT Unclassified/Unlimited, DTIC, NTIS			**12.B DISTRIBUTION CODE**
13. ABSTRACT (MAXIMUM 200 WORDS) Organizations that own or use software assets require a structured and validated approach for making decisions on how to update, migrate or reengineer their legacy assets. A model has recently been developed to understand technical transformations at different levels of abstraction. However, this model, which focuses on technical issues, is not yet accessible for decision-makers. This report outlines the foundation of a structured and coherent method, based on the "horseshoe" model, that will help practitioners make appropriate reengineering choices.			
14. SUBJECT TERMS architecture, architecture reconstruction, reengineering, reengineering decision aid			**15. NUMBER OF PAGES** 21 pp.
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	**18. SECURITY CLASSIFICATION OF THIS PAGE** UNCLASSIFIED	**19. SECURITY CLASSIFICATION OF ABSTRACT** UNCLASSIFIED	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102