A Conceptual Foundation for Software Re-engineering

Eric J. Byrne

Department of Computing and Information Sciences
Kansas State University, Manhattan, Kansas 66506

Abstract

This paper presents a conceptual foundation for software re-engineering. The foundation is composed of properties and principles that underlie re-engineering methods, and assumptions about re-engineering. The value of this conceptual foundation is its ability to model our understanding of re-engineering, how it is practiced, and how it can be practiced. A general model of software re-engineering is established, based on this foundation. This model, along with its underlying foundation, proves useful for examining re-engineering issues such as the re-engineering process and re-engineering strategies.

1 Introduction

Useful ideas and insights have emerged from research and experience in software re-engineering. Fundamental ideas that characterize re-engineering and underlie potential re-engineering methods are often loosely stated or implicitly assumed when discussing re-engineering. The intent of this paper is to consolidate and structure software re-engineering ideas into a coherent and useful framework for understanding and exploring re-engineering.

What is software re-engineering? Numerous definitions have been given [8, 12]. The most widely accepted definition is [5]:

> [Software] re-engineering, also known as both renovation and reclamation, is the examination and alteration of a software system to reconstitute it in a new form and the subsequent implementation of the new form.

A software system consists of one or more programs, data files, databases, and job control scripts. Any product of the software lifecycle is considered part of a system including designs, requirements, documents, and manuals. An entire system can be re-engineered or just portions.

Why is software re-engineered? A typical response is that re-engineering is done to transform an existing "bad" system into a new "good" system [12]. Many different reasons for re-engineering exist. Most reasons assume that an existing system needs to be improved. This is an assumption inherent to the concept of software re-engineering:

Assumption 1 *The re-engineering of a software system produces a new form of the system that is better, in some way, than the original form.*

The goal of software re-engineering is to take an existing system and generate from it a new system, called the target system, that has the same properties as a system created by modern software development methods. These desired software properties include: maintainability, portability, reliability, reusability, quality of documentation, testability, and usability. Empirical results support Assumption 1. Re-engineering a system can indeed result in improvements [11].

Modern software development methods strive to instill good properties into new software systems. The re-engineering challenge is how to instill these properties into an already existing system. Re-engineering research explores techniques to achieve this goal [7]. However, to develop successful techniques it is necessary to understand the re-engineering problem [6].

This paper presents a *conceptual foundation* for software re-engineering. This foundation is an attempt to identify underlying principles and assumptions and explain them in a way that furthers an understanding of re-engineering. This foundation provides a framework for gaining insights into re-engineering, understanding the re-engineering process, identifying the basis for effective re-engineering techniques, and characterizing and comparing different re-engineering methods.

0-8186-2980-0/92 $03.00 © 1992 IEEE

2 Levels of abstraction

Understanding how software is developed is useful for understanding how software can be re-engineered. The *concept of levels of abstraction* that underlies the development process also underlies the re-engineering process. This concept is used to model software development as a sequence of phases, where each phase corresponds to a particular **level of abstraction**. Figure 1 shows the levels of abstraction.

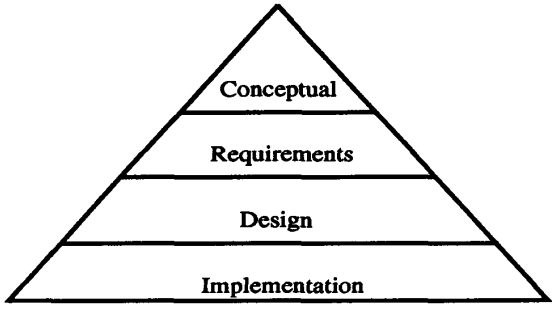

Figure 1: Levels of Abstraction

In software development, the purpose of each phase is to define certain system characteristics that determine the degree to which a system exhibits a particular property. Different phases define different characteristics. Each phase produces a system representation: a collection of information describing a system.

The **conceptual abstraction level** is the highest level of abstraction. Here, functional characteristics of a system are described in general terms. In the **requirement abstraction level** functional characteristics of a system are described in detailed terms. In these first two levels internal system details are not mentioned. In the **design abstraction level** system characteristics such as architectural structure, system components, interfaces between components, algorithmic procedures, and data structures are described. The **implementation abstraction level** is the lowest level. Here, a system description focuses on implementation characteristics, and is represented in a language understood by a computer.

Within an abstraction level are **degrees of abstraction**, that distinguish differences in abstraction within an abstraction level. For example, two degrees of abstraction within the design abstraction level are high-level design (i.e. expressing the architectural structure of a system) and detailed-design (i.e. expressing the internal structure of system components).

Each development phase produces a system representation and software is developed by a sequence of phases, thus a system can be described at several different abstraction levels. A system representation can be a requirements specification, module interface document, source code, data model, etc. This is an important property of the concept of levels of abstraction.

Property 1 (Separation of Concerns) *For each level of abstraction there exists a system representation that explicitly describes a system's characteristics. Each level of abstraction defines a different set of characteristics.*

The information held within a particular level or degree of abstraction is determined by the system characteristics of interest at that level or degree. The information held is further determined by the method used to create the information (i.e. functional decomposition vs. object-oriented design for designs), and the notation used to express the information (i.e Z or VDM vs. English for requirements).

In software development, information within a level or degree of abstraction is used to create information within the next lower abstraction level, creating a dependency between levels or degrees. This dependency establishes an ordering levels. Figure 1 reflects this ordering. Furthermore, this dependency only exists in one direction. Information within one level or degree does not influence information at higher levels or degrees. This is another important property.

Property 2 (Information Inclusion)

(a) *Information contained within a level of abstraction influences information contained at lower abstraction levels.*

(b) *Information contained within a level of abstraction has no influence on information contained at higher abstraction levels.*

Property 2 says that information within a particular level or degree of abstraction derives, in part, from information at higher levels or degrees. Generally, there is a one-to-many relationship between an item of information within one level and information at lower levels that derive from it. For example, one requirement can typically be traced to one or more components of a design. One design component is implemented by one or more lines of source code. As the level of abstraction decreases a system description becomes more detailed and the amount of information increases. (Hence, the pyramid shape used in Figure 1.) This is another important property.

Property 3 (Information Volume) *As the level of abstraction decreases, the amount of information describing a system increases.*

Since each development phase focuses on defining particular characteristics of a system, different system characteristics come into existence at different levels or degrees of abstraction. Functional characteristics are developed at the conceptual and requirements level. Architectural characteristics are developed at the design level, etc. This is another important property.

Property 4 (Creation of Characteristics) *Each characteristic of a system is created at a particular abstraction level, and the influence of each characteristic propagates downwards to lower abstraction levels.*

These properties of the concept of levels of abstraction contribute to our understanding of software re-engineering. Many assumptions underlying re-engineering can be described in terms of these properties. For example, software development starts with an idea for a system and creates a system representation for each abstraction level. Where as, re-engineering starts with an existing system representation. This is an important difference between development and re-engineering that is fundamental to re-engineering.

Assumption 2 *Software re-engineering begins with an existing system representation expressed at some level of abstraction.*

An important issue is how to identify the abstraction level at which re-engineering work should be conducted. Re-engineering changes the characteristics of a software system. To alter information about a system characteristic work at either the abstraction level at which the characteristic is introduced (Property 4) or any level below that (Property 2.a).

Typically, information about a system characteristic is explicitly expressed at the level or degree at which the characteristic is created. At lower levels or degrees information about a characteristic is frequently expressed implicitly, making the information difficult to discern and manipulate. For example, requirements are stated explicitly in a requirement specification, but only implicitly within a design. This problem also occurs in software maintenance. Experience leads to an important assumption underlying re-engineering.

Assumption 3 *To alter a system characteristic, work at the level of abstraction at which information about that characteristic is explicitly expressed.*

If Assumption 3 is followed, the likelihood of obtaining the best possible re-engineering result is increased. The phrase "best possible result" means that the target system has the desired system characteristics and properties. Further, the degree of bias in the target system introduced from the existing system can be reduced. Biases in a target system occur because a target system is derived from an existing system [3].

Assumption 4 *A system characteristic can be altered by working within a level of abstraction below the level at which information about the characteristic is explicitly expressed. However, the best re-engineering result might not be achieved.*

To manipulate characteristics related to the system source code work at the implementation abstraction level. To manipulate characteristics of the system architecture work at the design abstraction level. To manipulate functional characteristics work at the requirements abstraction level.

From Properties 2.b and 4 another assumption can be identified. Assumption 5 says that a characteristic can not be *altered*, in the sense of manipulating information about that characteristic, by working at a higher abstraction level that contains no information about that characteristic. However, by working at the higher abstraction level it is possible to *replace* all information about a characteristic. (See Section 5.)

Assumption 5 *A system characteristic can not be altered by working within a level of abstraction above the level at which information about the characteristic is introduced.*

The contribution of the concept of levels of abstraction is the idea that a system can be described by a hierarchy of representations. Each system representation corresponds to a particular level or degree of abstraction. Each level or degree emphasizes different system characteristics. Finally, the higher the abstraction level the less information about a system there is to comprehend.

3 Software re-engineering principles

Assumptions 3 and 4 state that to change a system characteristic it is best to work at an abstraction level where information about the characteristic is explicitly expressed. In practice, there exists only one system representation that accurately describes the

system. It is unusual to have accurate system representations at several different abstraction levels. Generally, a system's source code is the only up-to-date representation. Frequently, any requirement specifications or design documents, if they exist, have not been updated as the system evolved. This condition leads to an assumption that underlies many re-engineering methods.

Assumption 6 *For any existing software system, the only accurate and up-to-date system representation is its source code. Any existing requirement or design documentation can not be trusted to describe the current system accurately.*

What if the only accurate existing system representation is not from the level of abstraction for which information about the system characteristic to be changed is explicitly expressed? This question poses a fundamental challenge for software re-engineering. One solution is to reconstruct the system representation at the appropriate abstraction level. Methods for reconstruction are an area of active research [1, 2, 10].

There are three principles that underlie many re-engineering methods. These principles of software re-engineering are: Refinement, Abstraction, and an optional principle – Alteration. These principles, together with the properties of the concept of levels of abstraction, and the assumptions identified in this paper form a conceptual foundation for software re-engineering.

3.1 Refinement

Modern software development practices provide the ability to generate a system representation, at a particular level or degree of abstraction, from the next higher level or degree. Construction of a lower level abstraction is achieved by a series of **refinements**. The level of abstraction is decreased by replacing system information expressed at the current level with more detailed information expressed at the next lower level of abstraction. The choice of a particular refinement is guided by the development method used.

Principle of Refinement *The gradual decrease in the abstraction level of a system representation is caused by the successive replacement of existing system information with more detailed information.*

A software system is created by moving downward through the levels of abstraction. This downward movement is considered forward movement through the development process, hence software development is also called **forward engineering**.

3.2 Abstraction

When the only accurate existing system representation is not at the level of abstraction required by the planned system changes, it is useful to reconstruct the missing representation. Software re-engineering methods often assume that it is possible to reconstruct a system representation that is either missing or not up-to-date.

Assumption 7 *If a system representation at a particular abstraction level is missing or not up-to-date it is possible to, at least partially, reconstruct that representation.*

One means of reconstructing a system representation is to abstract information from an existing representation at the next lower abstraction level by using **abstraction**. Abstraction either ignores information items expressed within a representation or replaces several items with a single information item expressed at a higher abstraction level. A good abstraction is one in which information that is significant is emphasized, and details that are immaterial or diversionary are suppressed [9]. Abstraction is the reverse of refinement.

Principle of Abstraction *The gradual increase in the abstraction level of a system representation is created by the successive replacement of existing detailed information with information that is more abstract. Abstraction produces a representation that emphasizes certain system characteristics by suppressing information about others.*

Abstraction is an upward movement though the levels of abstraction. This upwards movement is called **reverse engineering**. Software reverse engineering is defined as the process of analyzing a software system to identify the system's components and their interrelationships and to create representations of the system in another form or at a higher level of abstraction [5].

The use of abstraction in re-engineering is based on Assumption 8.

Assumption 8 *A system representation at a particular level of abstraction can, at least partially, be reconstructed from an existing representation expressed at a lower level of abstraction by using the principle of abstraction.*

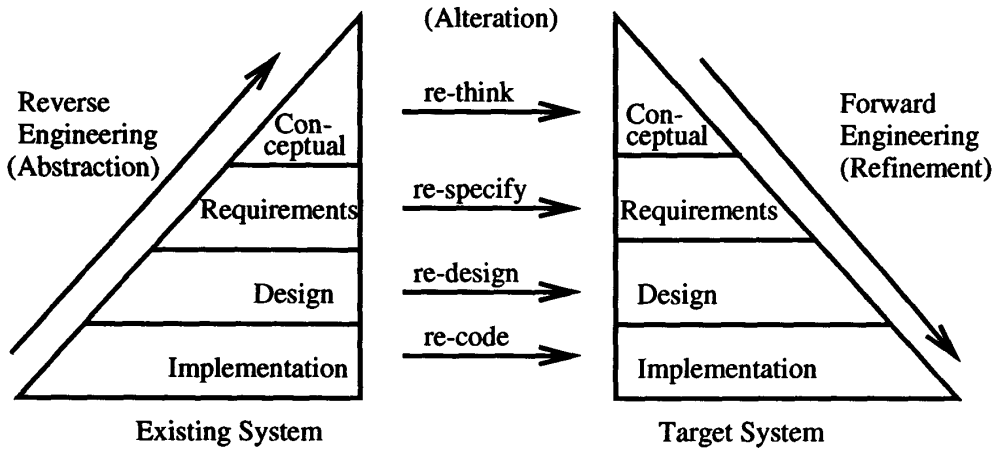

Figure 2: General Model for Software Re-engineering

Abstraction plays an important role in software re-engineering. If the only accurate description of a system is its source code (Assumption 6), abstraction can be used to produce a design or requirement level system representation (Assumption 8).

3.3 Alteration

Given a system representation at the appropriate abstraction level the ability to manipulate information about system characteristics is needed. System information can be changed by using **alteration**. Alteration manipulates information by including new information, deleting unwanted information, and changing existing information without changing the level or degree of abstraction.

Principle of Alteration *Alteration is the making of one or more changes to a system representation without changing the degree of abstraction. Alteration includes the addition, deletion, and modification of information.*

Alteration is not essential for re-engineering. Typically, alteration provides a bridge between abstraction and refinement. The use of alteration in a re-engineering project depends on the re-engineering strategy. (See Section 5.)

The software re-engineering paradigm commonly presented in the literature rests on three principles: Abstraction, Alteration, and Refinement. These principles, together with the properties of the concept of levels of abstraction, and the assumptions stated in this paper, form a conceptual foundation for software re-engineering.

4 Modeling software re-engineering

Use of the conceptual foundation ensures a consistent interpretation of software re-engineering properties and characteristics. In the remainder of this paper, the foundation is used to explore re-engineering. Two issues are examined in this section. First, the re-engineering process is analyzed using a model built on the foundation. Second, general classes of system changes are established. In the next section, strategies that underlie re-engineering methods are identified and discussed.

4.1 A general software re-engineering model

Figure 2 shows a general software re-engineering model. This model can be interpreted using the conceptual foundation. Properties 1 and 2 establish the abstraction levels and their ordering. The triangular shapes derive from Property 3. Assumptions 7 and 8 justify the use of the Principle of Abstraction, which is the basis for reverse engineering. Assumption 3 and the Principle of Alteration show the level of abstraction at which certain types of change can be made. The Principle of Refinement is the basis for forward engineering, which creates a target system implementation.

This model suggests that re-engineering begins with the existing system implementation and produces a target system implementation (Assumption 6). Variations of this model can be drawn by ignoring Assumption 6 and relying on Assumption 2 to suggest a starting level.

This model is useful for understanding technical aspects of the re-engineering process [4]. The sequence of reverse engineering, re-design (or re-code, or re-specify, or re-think), and forward engineering that is commonly mentioned in the literature clearly rests on the three re-engineering principles of Abstraction, Alteration, and Refinement. This sequence establishes a conceptual basis for the re-engineering process, as shown in Figure 3.

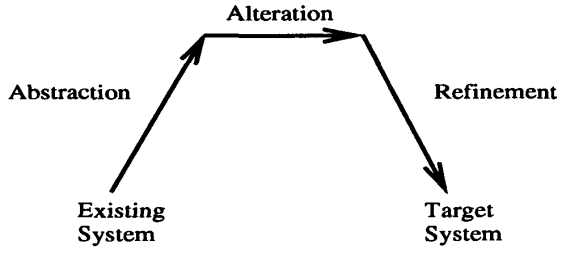

Figure 3: Conceptual Basis for the Re-engineering Process

Figure 3 demonstrates the use of the conceptual foundation for explaining re-engineering characteristics. This figure uses arrows to represent the re-engineering principles. An up-arrow represents Abstraction, a horizontal arrow represents Alteration, and a down-arrow represents Refinement. The arrows for Abstraction and Refinement are sloped suggesting the respective decrease and increase of system information (Property 3).

Figure 2 is an abstract model of the fundamental characteristics of the re-engineering process. Basic re-engineering steps are given. What is not modeled are process inputs, other than an existing system representation, and process outputs, other than a target system representation.

This general re-engineering model, based on the conceptual foundation, brings the individual foundation components together into a coherent form. The expressive power of the model is used by emphasizing the components of the conceptual foundation within the model.

4.2 Types of change

The model in Figure 2 suggests several possible paths from an existing system implementation to the target system implementation. The choice of a path depends partly on the system characteristic(s) to be changed (Assumption 3). System characteristics can be divided into groups. Characteristics within the same group are best changed by working within the same level of abstraction. In Figure 2 these groups are identified by the *type of change* necessary: re-code, re-design, re-specify, and re-think.

Re-code: changes to implementation characteristics. Language translation and control-flow restructuring are source code level changes. Other possible changes include conforming to coding standards, improving source code readability, renaming program items, etc.

Re-design: changes to design characteristics. Possible changes include restructuring a design architecture, altering a system's data model as incorporated in data structures, or a database, improvements to an algorithm, etc.

Re-specify: changes to requirement characteristics. This type of change can refer to changing only the form of existing requirements. For example, taking informal requirements expressed in English and generating a formal specification expressed in a formal language such as Z. This type of change can also refer to changing system requirements. Requirement changes include the addition of new requirements, or the deletion or alteration of existing requirements.

Re-think: changes to conceptual characteristics. This type of change can result in drastic changes to a system. Re-thinking a system means manipulating the concepts embodied in an existing system to create a new system that operates in a (slightly) different problem domain.

These different types of change were identified using the general re-engineering model. The underlying foundation provided the means to characterize these different types. This combination of model and foundation yields another observation. Changes within a particular group do not induce system changes at higher levels of abstraction (Property 2.b), but do result in changes to lower abstraction levels (Property 2.a).

These types of change can be modeled using components of the conceptual foundation. Figure 4 uses the re-engineering principles to form a conceptual basis for the types of change. This conceptual basis assumes that Assumption 6 holds. Notice that the

higher the level of abstraction at which Alteration occurs, the longer the arrows for Abstraction and Refinement. This increasing arrow length reflects the growing distance between the implementation abstraction level and the abstraction level at which alterations occur. The length of the arrow for Alteration differs for each type of change also. This figure illustrates the bridge that Alteration forms between Abstraction and Refinement. The higher the level of abstraction the shorter the bridge, i.e. the less alteration work to be done. (Follows from Property 3.)

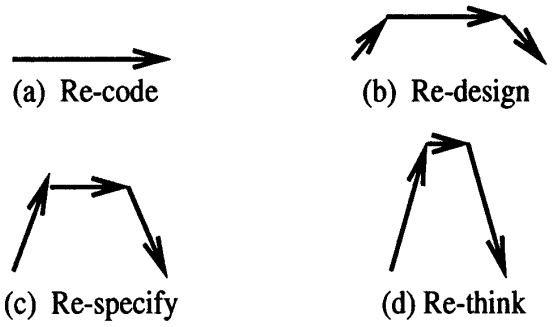

Figure 4: Conceptual Basis for Types of Change

5 Software re-engineering strategies

The abstract/alter/refine paradigm reflected in Figures 3 and 4 is but one possible strategy for re-engineering. Using the general re-engineering model and the conceptual foundation is it possible to identify several fundamental strategies. Three strategies are discussed in this section: Rewrite, Rework, and Replace. These strategies and the four types of change are then used to explain the re-engineering process in greater detail.

5.1 Strategies

A re-engineering strategy specifies the basic steps for a re-engineering method. Different strategies differ in their approach to re-engineering. These differences can be explained using the conceptual foundation. To identify re-engineering strategies observations are made by studying the general re-engineering model.

The first observation is that given a representation of an existing system, at some level of abstraction, it is possible to use just the Principle of Alteration to produce a representation of the target system at the same abstraction level. For example, it is possible to use just alteration on existing system source code to produce target system source code. This observation leads to the first re-engineering strategy.

Strategy 1 (Rewrite) *This strategy incorporates only the Principle of Alteration. To change an existing system, alteration is used to transform the existing system, represented at some level of abstraction, into the target system, represented at the same abstraction level.*

This strategy does a sideways transformation of information within a level or degree of abstraction. Figure 5(a) shows the conceptual basis for the Rewrite Strategy.

A disadvantage of this strategy is that changes are made regardless of whether the existing system representation is at the appropriate level of abstraction (Assumption 3). Depending on the type of change to be made and the starting system abstraction level (Assumption 2), the best possible re-engineering results might not be produced (Assumption 4).

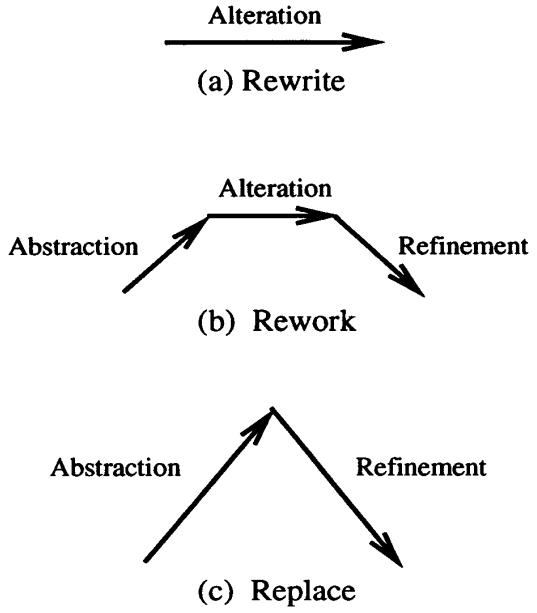

Figure 5: Conceptual Basis for Re-engineering Strategies

The second observation is that given an existing system representation, at any level or degree of abstraction except the uppermost conceptual degree, it

is possible to use the Principles of Abstraction, Alteration, and Refinement to produce a target system representation that achieves the best possible re-engineering result. This observation is more significant than the first. This observation says that it is possible to reconstruct the appropriate level or degree of abstraction at which to make system changes (Assumption 7), allowing the changes to be made at the appropriate abstraction level (Assumption 3). This observation leads to the second strategy.

Strategy 2 (Rework) *This strategy incorporates the Principles of Abstraction, Alteration, and Refinement. To change an existing system characteristic, abstraction is used to reconstruct a system representation at the appropriate abstraction level. Alteration is then used to transform the reconstructed system representation into the target system representation, at the same level of abstraction. Finally, refinement is used to create a suitable target system representation at a lower level of abstraction.*

A method based on this strategy can not be used if the starting abstraction level of the existing system representation is higher than the appropriate level for the planned changes. In practice, this will not be a problem. Figure 5(b) shows the conceptual basis for the Rework Strategy.

The third observation is that the Principle of Alteration is not essential. Given a representation of an existing system, at any level or degree of abstraction except the uppermost conceptual degree, it is possible to use only the Principles of Abstraction and Refinement to produce a target system representation that achieves the best possible re-engineering result.

To understand this observation consider a re-engineering project goal to restructure a program's control-flow, i.e. replace unstructured control flow constructs such as GOTO's with structured control flow constructs. A common method is to parse the source code and build a control-flow graph for the program (abstraction). A restructuring algorithm is applied to the control-flow graph producing a transformed control-flow graph with the property of being structured (alteration). The restructured control-flow graph is then translated back into the programming language (refinement). This method is based on the Rework Strategy.

Another method for control-flow restructuring is to use abstraction to reconstruct a program representation at a higher level of abstraction that contains no control-flow information (Property 4). Such a representation is at a higher degree of abstraction than a control-flow graph (Property 2.b). For example, the representation could describe the net effect of calling a routine, but give no information about control-flow within the routine. A target system representation at a lower level of abstraction is then produced by using refinement. This method results in the complete replacement of the program's control-flow. The basis for this method is the Replace Strategy.

Strategy 3 (Replace) *This strategy incorporates only the Principles of Abstraction and Refinement. To change an existing system characteristic, abstraction is used to reconstruct a system representation at a level of abstraction that contains no information about the characteristic. Refinement is then used to create a suitable target system representation at a lower level of abstraction. The system characteristic is created anew in the target system.*

An advantage of this strategy is that the replaced system characteristics do not derive from the existing system characteristics. The degree of bias introduced into the target system is reduced. Figure 5(c) shows the conceptual basis for the Replace Strategy.

5.2 Strategies and types of change

Software is re-engineered by following a process [4]. Different re-engineering projects follow different process variations. This process variability increases the difficulty of modeling the re-engineering process, because no process model applies to all possible re-engineering projects. The conceptual foundation can be used to identify the cause of this process variability. Consider three process factors: the abstraction level of the existing system representation, the type of change to be made, and the re-engineering strategy to be used.

Table 1 identifies possible re-engineering process variations. This table is read by asking: if a project starts with an existing system representation at abstraction level X, and plans to make Y type of change, can strategy Z be used?

Yes A suitable target system can be created (Assumption 3).

Yes[1] Same as **Yes**, but the starting degree of abstraction can not be the uppermost degree of abstraction within the conceptual abstraction level.

No It is not possible to start at abstraction level X, make Y type of change, and use strategy Z, because the starting abstraction level is higher than

Starting Abstraction Level	Type of Change	Re-engineering Strategy		
		Rewrite	Rework	Replace
Implementation Level	Re-code	Yes	Yes	Yes
	Re-design	Bad	Yes	Yes
	Re-specify	Bad	Yes	Yes
	Re-think	Bad	Yes[1]	Yes[1]
Design Level	Re-code	No	No	No
	Re-design	Yes	Yes	Yes
	Re-specify	Bad	Yes	Yes
	Re-think	Bad	Yes[1]	Yes[1]
Requirement Level	Re-code	No	No	No
	Re-design	No	No	No
	Re-specify	Yes	Yes	Yes
	Re-think	Bad	Yes[1]	Yes[1]
Conceptual Level	Re-code	No	No	No
	Re-design	No	No	No
	Re-specify	No	No	No
	Re-think	Yes	Yes[1]	Yes[1]

Table 1: Re-engineering Process Variations

the abstraction level required by the particular type of change (Assumption 5).

Bad A target system can be created, but the likelihood of achieving the best possible re-engineering result is reduced (Assumption 4).

Table 1 shows 30 workable re-engineering scenarios, 24 of which are likely to produce the best possible re-engineering product. The three factors used in this table account for much of the variability within the re-engineering process. This is a useful result that improves our understanding of the re-engineering process. This result shows the power and value of the conceptual foundation for studying software re-engineering.

6 Conclusion

This paper presented a conceptual foundation for software re-engineering. This foundation consists of a collection of properties, assumptions, and principles that are often either loosely stated or implied when discussing re-engineering methods and problems. By articulating these ideas a foundation is established that enables researchers and practitioners to clearly define, discuss, compare, and evaluate re-engineering problems and methods. The conceptual foundation is useful for understanding re-engineering and proposed solutions to re-engineering problems.

The basis for the conceptual foundation rests on the *concept of levels of abstraction* and its properties. The assumptions and re-engineering principles that form the foundation are defined in terms of these properties. The concept itself is the result of years of experience and research in software development.

To clarify re-engineering concepts this paper has attempted to identify and clearly state assumptions that shape our current understanding of re-engineering. These assumptions influence our perception of how software is and can be re-engineered. Understanding these assumptions furthers our ability to evaluate and compare different strategies. Assumptions 3 and 4 can be used to judge the potential effectiveness of a strategy in a given situation.

The conceptual foundation was used to explain a general model for software re-engineering. In turn, this general model was used to identify the general types of change made to systems during re-engineering work. The identification of these types of change derives from Property 1.

An import use of the conceptual foundation is to identify possible strategies for re-engineering. Software re-engineering strategies rest on three principles: Abstraction, Alteration, and Refinement. To develop effective re-engineering methods, it is necessary to understand the role these principles play. Three strategies were defined: Rewrite, Rework, and Replace. Other strategies also exist, and it is expected that new strategies will be developed in the future.

The general re-engineering model shows that variations in the re-engineering process exist. The conceptual foundation provides the means to identify and explain the factors that cause these variations. Three important factors are: the abstraction level of the existing system representation, the type of change to be made, and the re-engineering strategy to be used. This ability to explain process variations shows the usefulness of the conceptual foundation.

This conceptual foundation is an attempt to identify underlying principles and assumptions and explain them in a way that furthers the understanding of re-engineering. This foundation provides a framework for gaining insights into re-engineering, understanding the re-engineering process, identifying the basis for effective re-engineering methods, as well as characterizing and comparing different re-engineering methods. This foundation is not complete and is expected to evolve as research and experience in software re-engineering accumulates.

References

[1] Benedusi, P., Cimitile, A., De Carlini, U., "A Reverse Engineering Methodology To Reconstruct Hierarchical Data Flow Diagrams For Software Maintenance," *Conference on Software Maintenance*, Miami, Florida, October 16-19, 1989, pp. 180 - 189.

[2] Biggerstaff, Ted J., "Design Recovery for Maintenance and Reuse," *Computer*, Vol 22., No. 7, (July 1989), pp. 36 - 49.

[3] Byrne, Eric J., "Software Reverse Engineering: A Case Study," *Software - Practice and Experience*, Vol. 21, No. 12, (December 1991), pp. 1349 - 1364.

[4] Byrne, Eric J., Gustafson, David A., "A Software Re-engineering Process Model," *Conference on Computer Software & Applications* (COMPSAC), Chicago, Illinois, September 23-25, 1992.

[5] Chikofsky, Elliot J., Cross II, James H., "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software*, Vol. 7, No. 1, (January 1990), pp. 13 - 17.

[6] Colbrook, A., Smythe, C., Darlison, A., "Data Abstraction in a Software Re-engineering Reference Model," *Conference on Software Maintenance*, San Diego, CA, November 26-29, 1990, pp. 2 - 11.

[7] Haugh, James M., "A Survey of Technology Related To Software Re-engineering," *Second Annual Systems Reengineering Workshop*, March 25-27, 1991, Silver Spring, Maryland, Naval Surface Warfare Center, pp. 9 - 20.

[8] Moore, Tamra, Dumoulin, Terri, Trinh, My-Hanh, Hwang, Phillip Q., "Overview of Systems Reengineering Technology," *Second Annual Systems Reengineering Workshop*, March 25-27, 1991, Silver Spring, Maryland, Naval Surface Warfare Center, pp. 1 - 8.

[9] Shaw, M., "The Impact of Modelling and Abstraction Concerns on Modern Programming Languages," in Brodie, M.L., Mylopoulos, J., Schmidt, J.W. (Eds). *On Conceptual Modelling*, Springer-Verlag, 1984.

[10] Sneed, Harry M., Jandrasics, Gabor, "Inverse Transformations of Software From Code To Specification," *Conference on Software Maintenance*, Phoenix, Arizona, October 24-27, 1988, pp. 102 - 109.

[11] Sneed, Harry M., Kaposi, Agnes, "A Study on the Effect of Reengineering upon Software Maintainability," *Conference on Software Maintenance*, San Diego, CA, November 26-29, 1990, pp. 91 - 99.

[12] Yu, Don, "A View on Three R's (3Rs): Reuse, Reengineering, and Reverse-engineering," *ACM SIGSOFT Software Engineering Notes* , Vol. 16, No. 3, (July 1991), p. 69.