Presented at the 4th NSWC
Systems Reengineering Technology Workshop, February 8, 1994

Reengineering to Increase Maintainability and Enable Reuse

Grady H. Campbell, Jr.

Software Productivity Consortium
2214 Rock Hill Road
Herndon, Virginia 22070

As existing systems are changed to keep up with changing needs, their structure becomes less coherent and cohesive making it difficult and increasingly expensive to make further changes. In addition, as the legacy of complex automated systems grows while the available resources for upgrading or replacing them shrink, there is increasing concern for finding ways to leverage these systems as a base for new or improved existing systems. Reengineering is the concept of creating an improved system by judiciously reorganizing, revising, and extending an existing system. Reuse is a related concept in which a set of existing similar systems provide the basis for a product line of new systems. The Consortium's Synthesis methodology integrates reengineering within the framework of a systematic reuse-driven process that promises more cost-effective development and maintenance of software and systems in the future.

Motivations for reengineering

Reengineering is commonly viewed as a variant of system maintenance. Maintenance differs from other phases of system engineering in that its focus is an operational system that requires modifications to correct errors, to support customer needs more effectively, or to satisfy changed needs. Over its useful lifetime, a system must be repeatedly modified to stay responsive to the needs of the customers it is intended to serve. However, modifying a system in response to changing needs inevitably undermines the conceptual and structural integrity and subsequent modifiability of the system as needs continue to change. Reengineering is distinguished from other forms of maintenance because it presumes the need for a redesign of the existing system to make current and future changes more cost-effective and less error-prone. It is a type of maintenance because the system is not rebuilt from scratch but is derived in large part from the artifacts of the existing system.

This material is based in part upon work funded by the Virginia Center of Excellence for Software Reuse and Technology Transfer, sponsored by the Advanced Research Projects Agency under Grant # MDA972-92-J-1018. The content does not necessarily reflect the position or the policy of the U.S. Government, and no official endorsement should be inferred

Reengineering of a system involves first the analysis of the existing system, referred to as reverse engineering, and then reformulation, restructuring, and modification of the system so that required changes are easier to make reliably. Reengineering may be needed for several reasons:

- Documentation of the system's requirements, design, and implementation has either been lost or become unreliable because of subsequent changes to the implementation. Original needs may not be well understood. This makes it difficult and risky to change the system because of uncertainties in how parts of the system interact or why certain functions behave as they do.
- The needs served by the system have changed sufficiently that the original design is no longer a good solution. A redesign is required for the system to continue to be acceptable to its users.
- The technology upon which the system is based has become obsolete. To accommodate improved technology and better serve user needs, the system must be substantially redesigned.

In the worst case, a system may have all of these problems. Although reengineering can accomplish its intended purpose of creating a better structured, more maintainable system in the short run, it may do nothing to avoid recurrence of the problems that led to the need for reengineering in the first place. If recurrence of these problems is not somehow prevented by the reengineering or avoided in subsequent maintenance of the system, then after some time reengineering will again be necessary. Taking a different view of reengineering can reduce recurrence of these problems.

A framework for reengineering

The driving concern for reengineering is the ability to create a system that can be easily changed as customer needs change. The driving concern for reuse is the need to field multiple systems or system versions that satisfy similar yet differing needs, of one or several customers, without having to repeatedly develop similar software from scratch. In reality, these two concerns are the same:

the ability to produce similar systems, whether serially or concurrently, to satisfy similar needs.

Looking more closely at the possible motivations for reengineering a system reveals several alternative objectives:

- To make changes in the functioning or operational properties (e.g., reliability, performance) of an existing system
- To make it easier or safer to make current and future changes in an existing system
- To use existing systems as a foundation for similar future systems

When reengineering is motivated only by the first of these objectives, the situation is not particularly different from that of conventional development and maintenance. In this case, the objective is most likely addressed adequately by traditional approaches to maintenance in which the architecture of a system is upgraded or particular data structures or algorithms are replaced by improved alternatives. The other two objectives warrant a different approach based, the Consortium believes, on the concept of program families [1, 2].

When the objective of reengineering is either to make a system easier to change or to use legacy systems as a foundation for future systems, Dijkstra's and Parnas' concept of orienting development to a family of systems provides significant opportunities for leverage in comparison to a traditional, single-system orientation. Even a single system inevitably evolves through multiple versions because of poorly understood requirements or to accommodate changing requirements or technology. The Consortium's approach to reengineering is based on families of systems, within the framework of the Synthesis methodology for reuse-driven software processes [3]. Reengineering within a Synthesis process comprises conventional reverse engineering capabilities for the analysis of existing artifacts combined with an innovative reuse-driven approach to creating and using families of systems as a basis for both the development and maintenance of systems.

A reuse-driven software process

An organization's primary motivation for instituting a Synthesis process is that the organization perceives itself as having expertise in and serving the market for a cohesive business area. The market has the need for either a single evolving system or several similar systems, which in either case offers a basis for conceiving a family of systems.

The essence of our approach is that development should result in a family of similar systems from which it is possible to mechanically derive alternative members of the family for rapid delivery to customers. A family is not just an abstract conception but a concrete formulation. It is designed and constructed as the means for systematic production and modification of systems to satisfy diverse or changing needs.

A Synthesis process, as depicted in Figure 1, consists of two major activities: domain engineering and application engineering. These activities, described briefly here, are defined fully and in detail in [4], along with extensive practical guidance.

Application engineering is concerned entirely with the needs of a particular customer and with producing a system that effectively addresses those needs. Application engineering prototypically consists of four subactivities:

- Project management. Planning, monitoring, and controlling an application engineering project to respond to customer needs.
- Application modeling. Formalizing customer needs and analyzing alternative solutions in terms of a set of decisions that are sufficient to distinguish a particular instance of a supported family of systems.
- Application production. Producing a system by means of a prescribed mechanical selection, adaptation, and composition of reusable components, directed by the decisions made in application modeling.
- Delivery and operation support. Installing a system in its operational environment, training users, assisting them in effective system operation, and identifying changes that will make the system a better fit to customer needs.

Domain engineering focuses on how to make application engineering most effective in meeting both the objectives of the business and the needs of the targeted market. To achieve this, domain engineering formalizes a family of systems as a domain by identifying the common and varying features of the type of systems that the market requires. Typically, domain engineering supports multiple application engineering projects. Domain engineering consists of five subactivities:

- Domain management. The planning, monitoring, and control of the domain engineering effort. This encompasses coordination with application engineering project management and concern for all facets of process management including configuration management and quality assurance disciplines.

- Domain definition. Establishing the scope and extent of the domain and formalizing the variabilities that differentiate instances of the targeted family of systems.
- Product family engineering. Formalizing standardized (adaptable) requirements, design, and implementation for the family of systems and all associated deliverable and supporting work products.
- Process engineering. Formalizing a definition of a standardized application engineering process and creating automated support for its performance. The prototypical description of application engineering described above is tailored to suit the specific needs of the domain and associated projects.
- Project support. Assisting application engineering projects to make effective use of the domain. This includes validating whether the domain is responsive to project needs and identifying needed improvements and changes.

A domain is a formalization of a family of systems and an associated process for producing members of the family. A system is represented by a set of artifacts (i.e., work products). A system is not just a collection of implemented (i.e., code) components but includes associated requirements/design/user documentation, test materials, management plans, and any other artifacts that result from development or support the use or maintenance of the system. Synthesis is concerned with producing all of these when a system is needed. Creating a family of systems means creating a representation of each type of relevant artifact as a family in its own right. Furthermore, artifacts may be made up of components which are in turn instances of component families. An essential objective of a Synthesis process is to create a material representation of all necessary system, artifact, and component families.

The essence of a family in this sense is that it represent a set of 'similar' individuals, by which we mean individual things that are identical relative to a specified set of traits. A family is formulated as an abstraction that denotes a set of similar individuals and identifies the particular traits that determine membership in the family. Materially representing a family requires expressing not only the substance of similarity but, equally important, the details of variation (out of which come distinct individuals). Variations are additional traits that together make each of the individual members of a family unique and correspond to decisions that are necessarily deferred until a particular family member (i.e., individual system, artifact, or component) is needed. Production of an individual then reduces to resolving these deferred decisions as needed to designate and mechanically derive the corresponding member of the family.

Methods for creating and using component families are referred to as metaprogramming [5]. A metaprogramming technique specifies how to create a component family and subsequently transform it into concrete

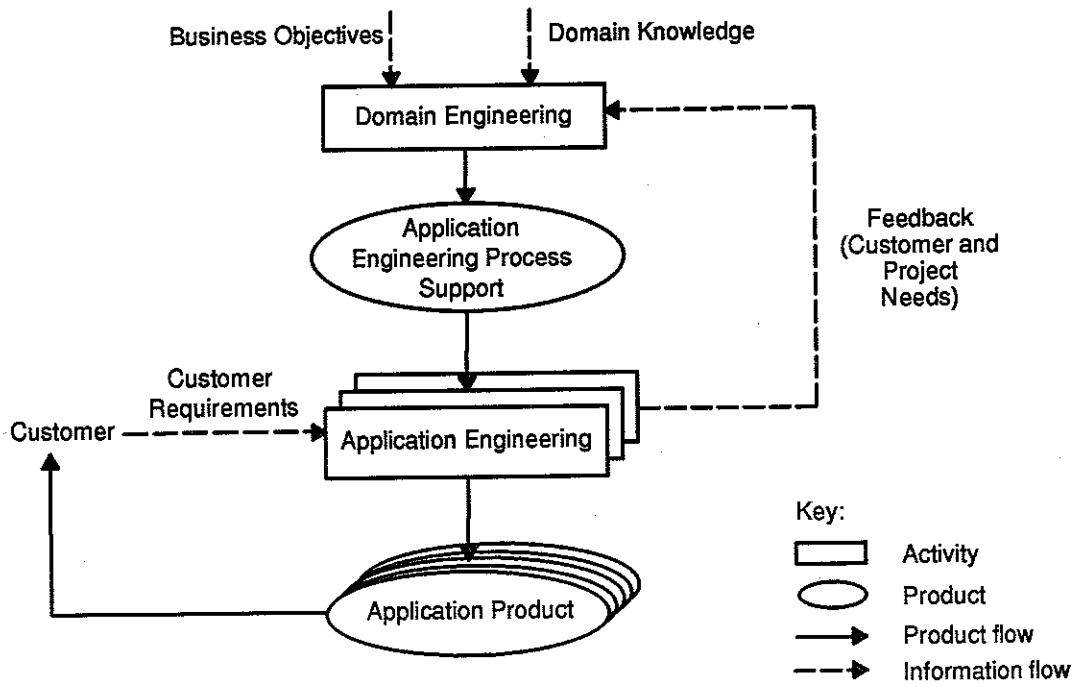

Figure 1. A Synthesis Process

instances. Mechanisms such as C preprocessor constructs, Ada generics, and form-letter capabilities of word processing software have proven sufficient for a viable Synthesis process. Other, special-purpose mechanisms, which are more complete but experimental, may provide additional leverage.

Experience with Synthesis

The Synthesis approach, until now emphasizing reuse with only limited concern for reengineering, has been used extensively by several industrial organizations. Two organizations, in particular, have contributed significantly to understanding Synthesis and how to achieve effective reuse:

- Rockwell Command and Control Systems Division. Rockwell began using Synthesis experimentally in 1990. They have now progressed to the point that they are evaluating its use in support of a substantial business area. Their experience is described in [6].
- Boeing/NAVAIR STARS* demonstration project. Boeing evaluated and selected the Synthesis methodology as the basis for its demonstration of megaprogramming and reuse [7]. This experience is now being transferred into trial use of Synthesis by the Naval Training Systems Command of NAVAIR.

In addition to these two examples, Synthesis is in experimental use on projects in Martin-Marietta, Lockheed, and other organizations. Based on this experience, Synthesis is proving to be a viable and sound approach for systematic reuse-driven software engineering. The experience so far in all of these is that a Synthesis process provides an effective capability for rapidly building multiple systems or system versions and subsequently modifying them as customer needs change. We believe that Synthesis also provides an effective framework for systematic reengineering as an aid to leveraging existing systems in producing new or improved software and systems.

Reengineering within a Synthesis process

An organization institutes a Synthesis process because it has expertise in a targeted business area and intends to serve the associated market. As a rule, requisite evidence of sufficient expertise to justify such a business commitment is that the organization has produced systems for this market in the past. Such legacy systems are a good initial source from which to create a domain as the formalization of a family. For effective use of legacy systems, reengineering is an integral element of the Synthesis process.

* STARS is the Software Technology for Adaptable Reliable Systems program of the Advanced Research Projects Agency.

Just as the emphasis in Synthesis is on creating a family of similar systems, the result of reengineering within Synthesis should be not just an 'improved' variant of the legacy system(s) but a family of similar systems from which alternative instances of the family can be derived. Derivable instances include alternative systems that are equivalent to an initial legacy system but improved in some way as well as systems that are useful hybrids or modifications of initial legacy systems.

Within Synthesis, reengineering is not viewed as a separate activity. Since a Synthesis process is meant to be a comprehensive engineering process, many of the necessary aspects of reengineering are already a part of the process. Currently, whenever a Synthesis activity involves the creation of a work product, it accommodates the analysis of existing systems as one source of the information in that work product. For example, requirements specifications of legacy systems can be a source for determining how best to express the requirements for the family as a whole. Similarly, test cases used in regression testing of those systems can be a source for creating test cases to be used in testing future systems. Only the use of reverse engineering capabilities need further elaboration as integral elements of Synthesis activities. In large part, this means the enhancement of the product family engineering activity of domain engineering to describe and explain the use of such capabilities.

A family of systems can be derived initially from either a single or several similar legacy systems. Reengineering may be concerned with any and all of the work products associated with a system. When a system is modified, changes are rarely limited to code components; reengineering should facilitate coordinated change across the entire set of artifacts associated with a system, including requirements, design, code, documentation, and test support.

One aspect of a Synthesis process is the design and implementation of component families. This takes the form of reengineering when components are available from legacy systems. Reengineering of legacy components to create a family can start with a bottom-up analysis of similarities among existing components. However, in a Synthesis process, analysis is guided by a top-down specification of component families based on an organization's business objectives. The challenges in creating a viable family by reengineering are to identify components that fit sufficiently within the scope of the envisioned family and to distinguish essential variations among identified components (i.e., driven by sound customer requirements or engineering concerns) from incidental (and therefore unneeded) variations. The leverage from this approach to reengineering comes from

recognizing that distinguishable instances can be derived from the unified abstraction of a family.

System reengineering as a generalization of software reengineering

System engineering is concerned with hardware, software, and manual procedures and the interactions among them. Much of the interest in reengineering has focused on software because of the increasing cost of maintenance associated with software changes. However, as defense spending shrinks, the useful life of individual systems grows longer. To respond to new and changing needs, there is a corresponding need and benefit in reengineering complete systems comprising hardware, software, and manual procedures. Reengineering a system can involve coordinated changes to any of:

- The system architecture, including physical and informational connections and the number, identity, and capabilities of the system's hardware and software components
- The design and implementations of individual hardware and software components
- The business/organizational and user processes within which the hardware/software system operates

Another dimension of reengineering at the system level, in contrast with the software level, is that the tradeoff between hardware, software, and manual procedures can be reconsidered. As technology advances, it becomes easier to move software functions into custom hardware. Alternatively, moving a hardware function into software can increase flexibility for modifying it in response to changing needs. Similarly, as system usage matures and manual procedures become more standardized, it becomes feasible to implement more of them in software.

As with software-oriented reengineering, the goal of system reengineering should not be narrowly to reconstruct a system to meet current needs but also to facilitate and reduce the costs of future changes as well. From this perspective, significant leverage arises from considering overall system concerns, as well as those related to each hardware, software, and procedural component of a system, within a reengineering approach. Furthermore, the similarity of motivations for reuse and reengineering justifies a unified approach for systems as well as for software.

Issues in reengineering

Most of the same issues that make system engineering a complex task, such as:

- Understanding the real requirements for a system so that effort is not wasted solving the wrong problem
- Evaluating alternative solutions and making engineering tradeoffs to attain a proper balance among system properties such as performance, reliability, development costs, and operating costs
- Verifying process performance and intermediate work products and validating the final product to ensure that the problem has been solved properly and correctly

are similarly a concern for reengineering. In addition to these common concerns, reengineering raises additional issues, specifically an extended verification problem and a deoptimization problem in reverse engineering. These problems are inherent to reengineering, regardless of approach.

Whenever a system is constructed, it must be validated to determine whether it satisfies the customer's actual needs. Because in the context of reengineering an operational system already exists, it is reasonable to expect that validation reduces to a problem of verifying the replacement system as the equivalent of the existing system. When the replacement system is supposed to have identical functionality to the current system, this equates to a total regression test of the replacement system. However, creating a replacement system with identical functionality is seldom feasible or necessary; creating only near-identical functionality is usually sufficient and less costly. Unfortunately, a divergence from identical functionality makes regression testing much more difficult. When, as is often the case because of changed needs, the replacement system also must differ in certain functionality from that of the current system, the problem takes on the characteristics to a greater or lesser degree of a complete revalidation. For reengineering to be practical, the effort of not only development but of verification and validation as well must be significantly reduced as compared to that of completing the system from scratch.

Reengineering generally requires the reverse engineering of a system for recovery of missing or obsolete design information or to establish precise, as-implemented requirements. Unfortunately, particularly in the case of software, the as-implemented structure is often an optimized equivalent of the intended design. For example, real-time embedded software usually entails responding to asynchronous events in the environment; logically, this corresponds to an architecture consisting of multiple concurrent processes. However, to satisfy stringent performance constraints, such an architecture has traditionally been implemented in a cyclic executive in which the logic of the processes is interleaved in a nonobvious fashion. Trivial reverse engineering would not reveal the

true logical structure of the software but instead would describe a much more complex linear structure. Reverse engineering techniques must be developed that help discover the original requirements and design while recognizing that the implementation is actually an optimization.

Conclusions

The Synthesis methodology for domain-specific software development offers a comprehensive framework for a reuse- and reengineering-based approach to revitalizing existing operational systems and producing new, more maintainable systems. Issues remain in the specific methods and technologies of reengineering, reuse-driven product lines, and process automation. However, based on extensive trial use by industry and government, the essential process framework is sound. Further work will demonstrate the benefits of taking such a product line perspective whether the motivation is to reduce the costs of new development or the costs of maintenance and whether the focus is on software or on systems.

Acknowledgments

Rich McCabe, Steve Wartik, and Roger Williams each provided helpful comments that greatly improved this paper.

References

[1] E.W. Dijkstra. "Notes on Structured Programming." *Structured Programming*, O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, Eds. Academic Press, London, 1972, pp. 1-82.

[2] David L. Parnas. "On the Design and Development of Program Families." *IEEE Trans. Software Eng.*, SE-2, 1976, pp. 1-9.

[3] Grady Campbell, Stuart Faulk, and David Weiss. *Introduction to Synthesis*, INTRO_SYNTHESIS_PROCESS-90019-N. Software Productivity Consortium, Herndon, Va., 1990.

[4] Software Productivity Consortium. *Reuse-Driven Software Processes Guidebook*, SPC-92019-CMC. Software Productivity Consortium, Herndon, Va., 1993.

[5] Grady Campbell. "Abstraction-Based Reuse Repositories." *AIAA Computers in Aerospace VII Conference*, Monterey, Ca., 1989, pp. 368-373.

[6] James O'Connor, Catharine Mansour, Jerry Turner-Harris, and Grady Campbell. *Exploring Systematic Reuse for Command and Control Systems*, SPC-92020-CMC. Software Productivity Consortium, Herndon, Va., 1993.

[7] B. Freemon. *STARS PSA S01 Experience Report*, D495-20154-1. The Boeing Company. 1993.