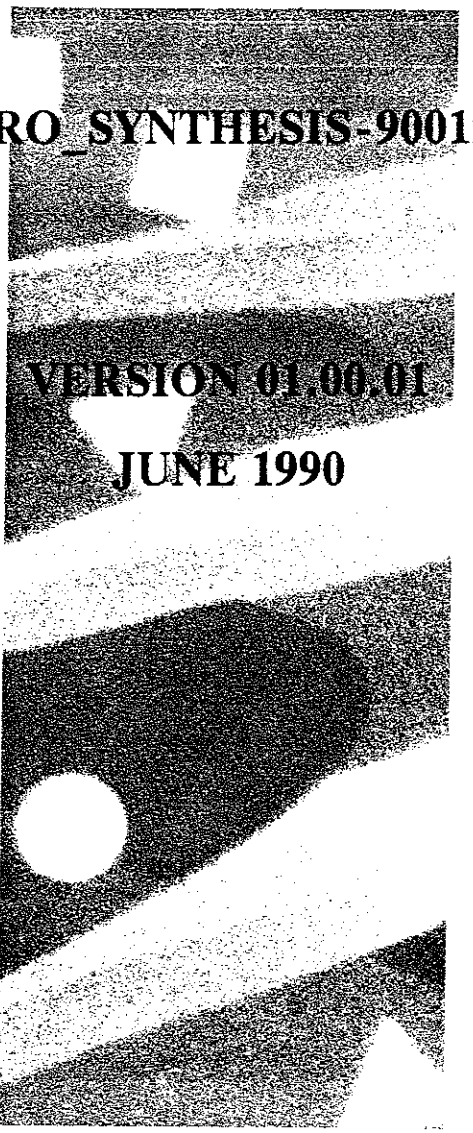
# Introduction to Synthesis



**INTRO_SYNTHESIS-90019-N**

**VERSION 01.00.01**

**JUNE 1990**

Software Productivity Consortium

# Introduction to Synthesis

## INTRO_SYNTHESIS-90019-N

## VERSION 01.00.01

## JUNE 1990

**Grady H. Campbell, Jr.**
**Stuart R. Faulk**
**David M. Weiss**



SOFTWARE PRODUCTIVITY CONSORTIUM
SPC Building
2214 Rock Hill Road
Herndon, Virginia 22070

Autocode and Matrix X are registered trademarks of Integrated Systems, Inc.

Mathematica is a trademark of Wolfram Research Inc.

MetaTool is a registered trademark of AT&T.

NeXtStep is a trademark of Next, Inc.

Refine is a trademark of Reasoning Systems Inc.

Spectrum is a trademark of Software Architecture & Engineering.

TAE is a registered trademark and service mark of NASA.

TEDIUM is a registered trademark of Tedious Enterprises, Inc.

# CONTENTS

# FIGURES

*This page intentionally left blank.*

# PREFACE

This is the first in a series of reports about the Synthesis approach to software production that is being developed by the Software Productivity Consortium. These reports are a product of the Synthesis Project, created in January 1990 to develop, validate, and communicate the principles of a Synthesis methodology. This report is an introduction for those unfamiliar with the Synthesis concept. It explains what Synthesis is, with only brief descriptions of how the Consortium expects to achieve it. Later reports in the series will explain the operational concept for Synthesis, compare Synthesis with other similar work, and present a detailed economic model for Synthesis processes. When taken together, these reports form a systems analysis for Synthesis. They explain the underlying motivation, the tradeoffs that were made, the economic rationale, and the technological decisions that have gone into the development of Synthesis.

*This page intentionally left blank.*

# ACKNOWLEDGEMENTS

A number of people have contributed to the vision of Synthesis presented here. All of the members of the Consortium's Synthesis Project have been vital to filling in the Synthesis vision and bringing the process closer to a usable approach. In particular, the case study by the Design Composer team—Neil Burkhard, Jeff Facemire, James Kirby, Jr., and James O'Connor—has been instrumental in clarifying the vision. Skip Osborne has contributed by making the Synthesis process tangible through visual prototypes, and James Kirby, Jr. has contributed to the many discussions of applicable methods.

This report has benefited substantially from the thoughtful reviews provided by Paul Clements, Guy Cox, Ted Davis, Jerry Decker, James Guy, Rich McCabe, Ken Nidiffer, Art Pyster, Sam Redwine, and Steve Wartik.

*This page intentionally left blank.*

# EXECUTIVE SUMMARY

A major purpose of the Software Productivity Consortium is to investigate ways to improve significantly the software productivity of its member companies. Software productivity and quality problems have become a major risk factor in all industries trying to exploit advanced technology. Member companies have confirmed a variety of problems that characterize current practice. The Consortium's goal is to provide a means to achieve major member company objectives:

- Manageable requirements that are complete, understood, and unambiguous, and a process that can accommodate requirements changes.
- Methods/toolsets that integrate well with other methods and tools.
- Early system verification that detects errors early in development and keeps maintenance costs down.
- Methods and tools that make reuse effective.

An early premise was that significant productivity improvement could result only from a revised software development process, particularly one that facilitated prototyping, systematic reuse, and model-based analyses. This premise has evolved into a characterization of a new methodology of software development called *Synthesis*.

Synthesis is a methodology for constructing software systems as instances of a family of systems having similar descriptions. Its distinguishing features are:

- Formalization of domains as families of systems that share many common features but also vary in well-defined ways.
- System building reduced to resolution of requirements, and engineering decisions representing the variations characteristic of a domain.
- Reuse of software artifacts through automated adaptation of system components to implement engineering decisions.
- Model-based analyses of described systems to help understand the implications of system-building decisions and evaluate alternatives.

The key principle of Synthesis is that software systems can be categorized into families. A family comprises a set of systems with similar requirements definitions that are satisfied by a common software architecture, and a set of closely related design choices at the detailed level. Such a family of systems is potentially viable as a cohesive business area. A *domain* is the application area corresponding to such

a family of systems. By focusing requirements analysis on identifying relations and variations among a set of potential applications, the design can be standardized so that it is easily adaptable to those variations. The resulting application development process, called *application engineering*, consists entirely of refining and evaluating engineering decisions and the automated generation of software products. The process is applicable both to the initial generation of a system and to subsequent variations throughout its existence.

A new, supporting process called *domain engineering* makes a simplified application engineering process possible. Domain engineering is a systematic process for creating an environment for application engineering. Its purposes are: 1) to develop a standard form for representing requirements of systems in the domain, and 2) to develop standard designs that can be mechanically adapted to accommodate requirements changes. A change to requirements represented in the standard form can then be implemented automatically by a design adaptation. The process is viable because it depends only on expertise already present in member companies. Current technology, with small extensions, is sufficient for the practice of Synthesis. The essential change is in the way engineers and managers look at the systems to be constructed. By viewing a system in the context of a family of similar systems, prototyping, reuse, and model-based analyses become the standard means of solving a problem in a framework tailored for a particular domain.

Success with Synthesis-like approaches can be found in application generator programs. This approach is also analogous to CAD/CAM systems used in automated manufacturing. Such programs are successful because they automate the creation of solutions to a chosen class of problems entirely from user descriptions of problem requirements. The intent of Synthesis is to define a systematic, repeatable process for creating such generators for families of programs in business areas of interest to the member companies.

Synthesis has five key benefits for member companies that address the difficulties characterizing their current practices:

- ***Capture and Leveraging of Expertise.*** Synthesis aids the capture of software engineering and domain expertise for the production of a set of similar systems. Such expertise is a major corporate asset that is now easily lost through personnel turnover.

- ***Productivity and Quality.*** The results of domain engineering are shared among projects and across system development/maintenance iterations. This substantially reduces the effort for system building and the potential for introducing errors. Subsequent enhancements in domain support and error correction are shared.

- ***Ease of Change.*** Since the explicit form of system representation is in application terms, traceability to customer requirements is direct. Since all requirements changes are handled as changes to an application model, no effort need be spent on understanding how previous requirements were translated into a design and implementation.

- ***Manageability.*** The ability to quickly generate a prototype system provides for better risk management. Major weaknesses in requirements understanding or existing solutions can be identified early. Domain engineering controls how much variance is allowed in application engineering practices. Preferred development methods and standards can be supported and enforced through domain engineering products.

- ***Customer Involvement.*** Representing the system solution in a domain-specific form allows for earlier customer review and feedback. Rapid creation of products from the represented solution allows earlier customer evaluation. Earlier feedback allows more rapid iteration to correct results.

The general form of a Synthesis methodology is adequately understood. Work continues in applying the methodology to problems in member company domains, and in developing a detailed written description of the process. The emphasis is on a definitive process that can be adopted incrementally and tailored to the specific requirements of individual member company projects. Some advances in methods and technology are needed, particularly for producing larger and more complex real-time systems. Key obstacles to the practice of Synthesis include implementing strategies for transition from current member company practices, and gaining customer acceptance and support. Although the costs and risks of accepting a new methodology should not be underestimated, there are significant, competitive risks in not broadly addressing the problems that result from current practices. The Synthesis approach will address these problems.

*This page intentionally left blank.*

# 1. INTRODUCTION

The potential for significant improvements in software productivity and quality exists with current technology. The problem is how to reorganize the software production process to eliminate unneeded rework. The cause of such rework is that engineers repeat the complete development cycle with each new product rather than taking advantage of work done in previous developments. Just as in many engineering disciplines, the solution lies in viewing system production as creating different members of a family rather than creating a new system each time requirements change. Once the basis for creating family members is established, the engineer can concentrate on determining the requirements for particular family members. This frees him from redundant design and implementation work. Particular family members can be produced rapidly, allowing more frequent customer validation. *Synthesis* is the approach for realizing such a software production process.

Features of rapid prototyping, reuse, and design for change are combined to achieve the Synthesis process. Synthesis can be applied whether the family is built as a series of applications under different projects with different contracts, or as a single system with many versions produced during a long maintenance cycle. In either case, product similarities allow them to be viewed as a family.

Surveys of the member companies confirm the problems associated with every phase of the development process that characterize current practice. Foremost among these are:

- Requirements that are incomplete, misunderstood, poorly defined, and changeable in ways that are difficult to manage.
- Methods and toolsets that fail to integrate smoothly across the system life cycle.
- Inadequate methods for verification of software systems, including error detection early in the life cycle.
- Inadequate methods and tools for effective reuse.

Introducing automated support for existing practices does not solve these problems. Current CASE tools typically offer support for representing, storing, tracking, and changing the products of the development process, and cannot provide more than incremental benefit. The common assumption underlying these tools is that software will continue to be developed using some minor variation on a conventional development model; i.e., requirements specification, design, code, and test. Such an approach may reduce the administrative burdens of software development, but does nothing to reduce the substantive work of conceiving and verifying adequate solutions to customer requirements.

The difficult part of the process is in creating the right program structures and relationships in the first place, and in verifying that they are, indeed, the right ones. Compared to the effort for creating

and verifying these conceptual structures, the effort to represent them is marginal. Similarly, various improvements in methods (e.g., object-oriented design) and programming languages (e.g., Ada) offer refinements on current practice. While these help remove some of the difficulties in representing and implementing the conceptual structures that comprise a software system, the essential difficulties in their construction and verification remain (Brooks 1987).

The Synthesis process focuses on establishing a new development model that more directly addresses the essential difficulties of software creation. It focuses on developing conceptual structures explicitly for reuse, then reuses rather than recreates them in subsequent developments.

Repeatedly solving the same problems with slight variations wastes effort. The conventional development model treats each product as if it were unique. It creates all of the life-cycle products from scratch. Synthesis replaces redundant work with reuse. Each software product is considered to be one of a set of related products. Whether starting a new project or maintaining an existing system, Synthesis views the problem as one member of a family of problems, then seeks solutions that can be adapted easily to solve any of the related problems. This allows available expertise to be leveraged across a set of recurring problems and removes the redundant effort of creating a unique solution for each problem.

A conventional development model also wastes effort in recreating the conceptual structures of a system across different representations of the same problem. Each phase of the development process produces another representation (possibly more refined) of the same information in a different form. The requirements specification, design document, code, and user documentation are intended to represent the same system from a slightly different perspective. For these documents to be useful references throughout the life of the system, they must be consistent with one another. This consistency must be maintained in the face of subsequent changes.

Many productivity and quality problems are traceable to the effort required to produce and maintain these disparate descriptions. The critical resource becomes the knowledge and ability of developers to translate between the various perspectives. Attempts to solve the problems by limiting change reduce system acceptability: changes needed to add capabilities and to improve ease of use cannot be made. Since current CASE tools are typically not integrated across the life cycle, they do little to alleviate the problem.

Synthesis eliminates the need to create and maintain redundant representations of a system. It mechanizes much of the translation process. Synthesis focuses on a single description of the solution to a problem, and promotes the rapid production of trial solutions. This description is called an *application model.* It is a representation of the application in terms of the application's requirements. Representations of design and implementation are excluded from it. Subsequent implementation of the requirements is based on a mechanical process of adapting and composing reusable components, both for the system code and documentation. The family of applications generated from these components and composition rules is a *domain*.

The rest of this report gives an overview of Synthesis in its current state of development, and offers evidence supporting the belief that the process will be effective in member companies. Future reports will provide more technical detail, compare Synthesis to current products and other work, and describe the results of ongoing Consortium case studies in applying a Synthesis approach.

# 2. WHAT IS SYNTHESIS?

While CASE tools have had little impact on software developers, there are notable exceptions among *application generators* (Watts 1987). Three characteristics common to application generators are particularly relevant to Synthesis:

- The user represents his requirements in terms of the problem (application-dependent terms), not in terms of the solution (design and program constructs).
- From the user's viewpoint, the requirements specification directly produces the software that implements the required functions.
- Similar requirements descriptions result in similar programs being generated.

For instance, mathematical problems may be represented as equations (e.g., Mathematica [Wolfram 1988]) rather than algorithms. User interfaces may be represented as available buttons and menus (e.g., TAE+ [Szcur 1989] and NextStep [NEXT, Inc.]) rather than controllers and interrupt routines. Transaction processing software may be represented as information needed rather than data structures and relations (e.g., TEDIUM [Blum 1988]). A flight control system may be represented as filters and gates (e.g., GALA [Coron 1988]) rather than code.

An application generator reduces software development to a rapid cycle of requirements definition and refinement. Requirements are always expressed in terms of the application itself. Since output of the process is executable application code, the design and coding phases are effectively eliminated. Validation, rather than verification, becomes a focal point for development.

Increased productivity in the application development cycle is achieved at the cost of analyzing applications the tool will construct, building a common design, and providing the mechanisms that underlie product generation. However, recurring costs for generating target applications are a fraction of the usual cost, and the initial investment is amortized over the set of systems constructed.

Application generators provide valuable insight into nonstandard development models that avoid many of the problems of more conventional models. For sufficiently constrained application domains, this can short circuit the development process for families of related programs, and substantially improve software productivity. The goal of Synthesis is to gain the same benefits in application areas common to member companies.

## 2.1 A DEFINITION

Synthesis is any methodology for constructing software systems as instances of a family of systems having similar descriptions. Its distinguishing features are:

- Formalization of domains as families of systems that both share many common features and also vary in well-defined ways.

- Adaptive reuse of software artifacts (in the sense that Ada's **generic** mechanism allows the adaptation of reusable parts to specific applications by instantiating parameters).

- Reduction of system building to resolution of necessary requirements and engineering decisions.

- Model-based analyses of applications to help understand the implications of system-building decisions.

While experience with application generators suggests how productivity might be improved, there is little guidance on how to produce reusable systems reliably. Current application generation tools are narrowly focused and restricted to highly constrained domains with properties readily favoring simple adaptations. Further, there is no established software engineering process for producing such reusable systems or incorporating reusable components in subsequent developments.

Attaining the potential benefits of Synthesis requires the definition of a software engineering methodology based on Synthesis. This process must be practiced as a well-defined and repeatable process in the sense that software engineering practices are characterized by the Software Engineering Institute's Assessment program (Humphrey 1989). In particular, it requires:

- A systematic process for constructing reusable system code and other components for a family of applications.

- A systematic process for using reusable components to create a new family member based on a statement of requirements.

- Evidence that such a process can be effectively applied in member company domains (both technically and contractually).

Developing a methodology that satisfies these criteria is the immediate goal of the Synthesis project. While there are potentially many such methodologies, the key is to define and apply one that can be practiced using available technology and sound software engineering practices. The Consortium's process is more correctly *a* Synthesis methodology, not *the* Synthesis methodology. Principles, tools, and techniques available today for specifying requirements for families, for designing for change, and for mechanically adapting parts make such a methodology feasible. Synthesis is a methodology based on these techniques. A Synthesis methodology consists of two related processes:

- ***Application engineering.*** Application engineering is a process for rapidly creating production quality software (including documentation and code) from reusable components. The reusable components allow mechanical generation of life-cycle products to reflect requirements decisions. The set of components, and the mechanisms supporting their reuse, are the *application engineering environment.* Generation does not need to be automated to be beneficial, although the mechanical nature of the process makes automation feasible. The application engineering process is analogous to the process of using an application generator to produce and validate a particular application instance.

- *Domain engineering*. Domain engineering is a process for creating an application engineering environment for a family of similar systems. Domain engineering includes all of the activities associated with identifying a target family of applications, capturing their variation, constructing an adaptable design, and defining the mechanisms for translating requirements into systems created from reusable components. Its similarity to application generators is in the process for creating the generator itself. Domain engineering differs in that it defines a repeatable process that can be applied by member companies to their areas of business.

### 2.1.1 Expected Benefits

There are five key benefits from Synthesis for member companies:

- *Capture and Leveraging of Expertise*. Synthesis aids the capture of software engineering and domain expertise for the production of a set of similar systems. Such expertise is a major corporate asset that is now easily lost through personnel turnover.

- *Productivity and Quality.* The results of domain engineering are shared among projects and across system development/maintenance iterations. This substantially reduces the effort for system building and the potential for introducing errors. Subsequent enhancements in domain support and error correction are shared.

- *Ease of Change*. Since the explicit form of system representation is in application terms, traceability to customer requirements is direct. Since all requirements changes are handled as changes to an application model, no effort need be spent on understanding how previous requirements were translated into a design and implementation.

- *Manageability.* The ability to quickly generate a prototype system provides for better risk management. Major weaknesses in requirements understanding or existing solutions can be identified early. Domain engineering controls how much variance is allowed in application engineering practices. Preferred development methods and standards can be supported and enforced through domain engineering products.

- *Customer Involvement*. Representing the system solution in a domain-specific form allows for earlier customer review and feedback. Rapid creation of products from the represented solution allows earlier customer evaluation. Earlier feedback allows more rapid iteration to correct results.

### 2.1.2 Addressing Member Company Problems

Synthesis addresses each of the primary member company problems identified earlier:

- *Requirements.* Focusing the application engineer on application modeling and rapid validation leads to better understood, better defined, and more complete requirements. Requirements changes are incorporated in only one description of a system, and can be rapidly implemented.

- *Integration.* A key precept of Synthesis is the systematic creation of an integrated solution to the construction of systems.

- ***Verification.*** As with most reuse models, Synthesis leverages the effort spent on verification of reused parts across all applications that use the parts.

- ***Reuse.*** Synthesis is an innovative approach to reuse that emphasizes designing components for prerun-time adaptation to multiple anticipated uses.

# 3. THE SYNTHESIS PROCESS

The Synthesis process fosters the development of application systems through the use of two closely related subprocesses: domain engineering and application engineering. Application engineering produces applications that meet customer requirements. Domain engineering produces the environment used to do application engineering.

Domain engineers produce a domain model that characterizes the application family. The domain model determines the characteristics of the application engineering environment that is built. Decisions made by the domain engineers in creating the model limit the choices that can later be made by application engineers. Engineers using the application engineering environment provide feedback to the domain engineers about how those limits affect the applications that they must build, i.e., feedback about their customers' needs. The domain engineers use the feedback to evolve the domain model and the application engineering environment. The application engineers also provide feedback about the deficiencies in the environment; i.e., what the environment needs to make application production more efficient. Figure 1 shows the Synthesis process, including the feedback loops between application engineering and domain engineering.

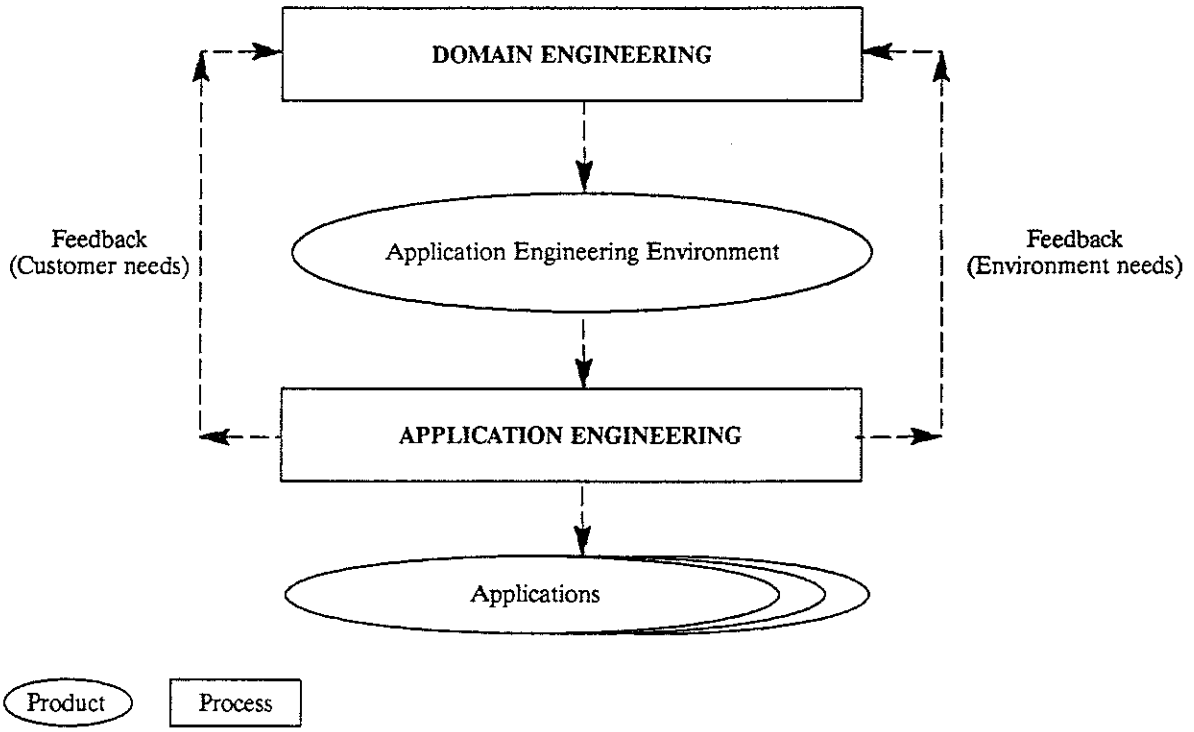



Figure 1. The Synthesis Process

As requirements and technology change and systems mature, the application engineering environment evolves. This happens both for different systems within a domain and for single, long-enduring systems that evolve through many versions. One evolutionary pattern is for environments to pace the growth of systems as they move from advanced development through full scale engineering development, production, deployment, and post-deployment support.

The remainder of this discussion uses as an example the Host-At-Sea Buoy System. Briefly, HAS buoys drift at sea, sample the environment (e.g., water temperature), determine their location through satellite or inertial navigation systems, and transmit reports on environmental conditions both at regular intervals and on demand to passing ships. To aid emergency sea rescue operations, the buoys may have emergency devices, such as flashing red lights, whose operation overrides other buoy activities. The domain engineering process for buoys determines limits on systems in the buoy domain by limiting the choices available to application engineers who develop buoy software. For example, the domain engineers determine ranges of configurations for buoys: sets of communications channels used on a buoy, types of sensors, and possible time interval between regular buoy transmissions. The environment produced as a result allows application engineers to make sets of choices within the limits imposed by the domain engineers; e.g., selecting three water temperature sensors, four air temperature sensors suitable for warm water operations, satellite navigation, reporting via link 11, and a reporting interval of four seconds.

## 3.1 SYNTHESIS AND SYSTEM DEVELOPMENT

A key assumption of Synthesis is that systems can be categorized into families with similar software requirements. Domain engineers must understand both the hardware and software system designs to characterize a domain. Under Synthesis, system requirements for family members (as characterized by the System Specification and System/Segment Specification) are input to the domain engineering process. A domain engineer must understand the results that system engineering produces well enough to forecast variations in system requirements. As systems in the domain evolve, so must his analysis of the domain, incorporating the results of systems engineering over the domain. As a result, domain engineering and its products provide a basis for communication between system engineering and software engineering.

Mature application engineering environments help formulate system requirements as those requirements evolve. An application engineering environment can evolve to meet the needs of a new system as the system's operational concepts and requirements become clear during its development. The environment may continue to evolve throughout the system's lifetime, especially for long-lived systems. In this case, the different versions of the system are the members of the application family. The feedback loop that carries information about customer needs from application engineering to domain engineering guides system evolution (see Figure 1).

A typical evolutionary cycle starts with an application engineering environment for a relevant family and a preliminary system specification for a new family member. The environment helps to define a refined system specification and mechanically produce preliminary Software Requirements Specifications, preliminary Software Design Documents, and other documentation. This process leads to requirements for evolution of the environment as well. Figure 2 shows such a scenario for evolving an application engineering environment and a new system together. System specifications and software documentation evolve from advanced development to full scale engineering development to production. At the same time, an application engineering environment evolves from an environment in

which minimally useful family members can be automatically generated, to a production environment in which all or most of the system software can be generated. (This discussion assumes that the system is composed of several Computer Software Configuration Items [CSCIs], possibly with a different application engineering environment used to generate software for each.)

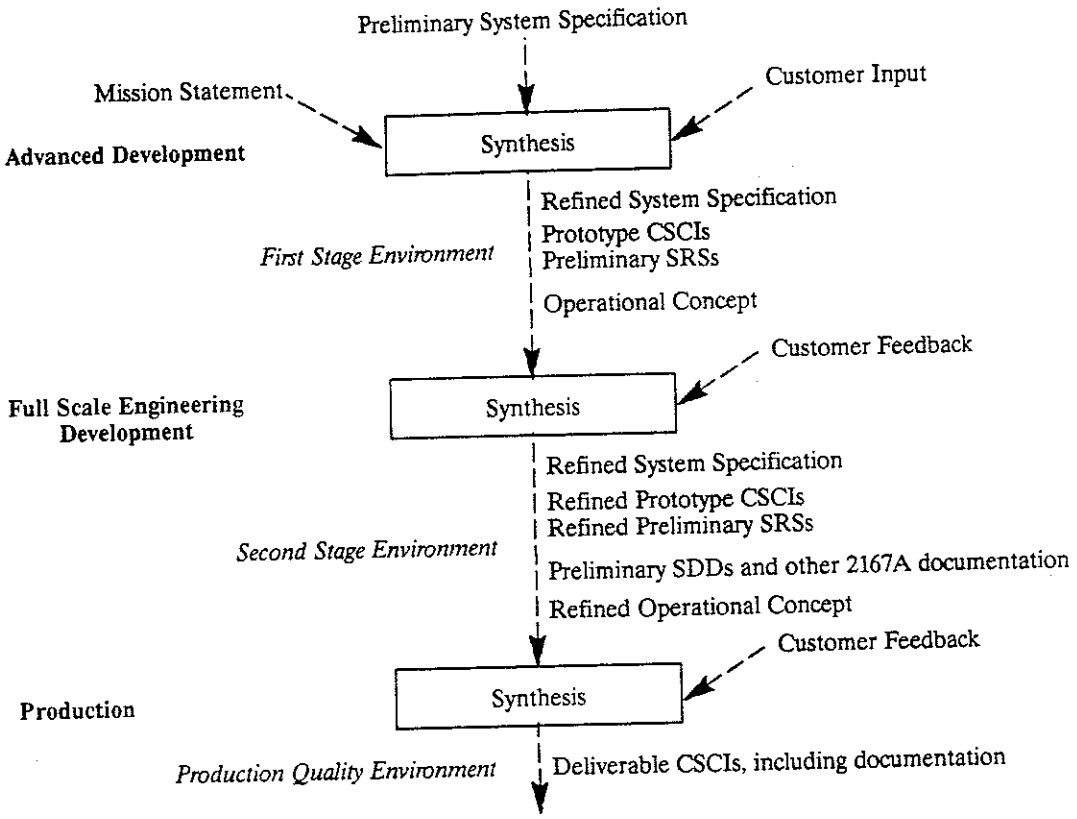



Figure 2. Evolution of Application Engineering Environment During System Development

## 3.2 SCOPE OF THE SYNTHESIS PROCESS

When Synthesis is used as the development process for all CSCIs in a system, it starts during systems analysis and is the primary process for software development. During the early stages of development, the domain model and the environment may automatically generate only minimally useful family members. Manual generation of some code and documentation may be necessary to complete family members. As the environment evolves, its capacity to generate family members increases, driven by new system engineering requirements and feedback from application engineers. Evolution may result from incorporating an enhanced domain model into the environment (i.e., a model that incorporates new domain requirements), or from adding new generational capabilities to the environment (e.g., enriching the supply of parts that the environment uses to generate code and documentation).

For example, the buoy application engineering environment may not include sonar sensors, but requires them for some versions of the buoy. The application engineer may ask the domain engineer to incorporate provision for such sensors in the environment. The domain engineer may add parts to the environment that can be used to control sonar. He can take advantage of the mechanisms already in the environment for generating code that samples sensors, stores sensor data, transmits

reports, or performs other buoy functions. Such an enhancement of the environment may be done with little or no change to the domain model.

Nonetheless, some members of the application family may have specialized requirements for which extending the domain model or enhancing the application engineering environment is not worthwhile. For such cases, the environment may generate most of the software, including documentation, for the application. The rest is produced outside the environment. As an example, a member of the buoy family might be outfitted with an oil detector to help clean up oil spills. Since only a few buoys need to have such a capability, it would not be worth the effort to extend the buoy environment to generate the code and documentation for such sensors. The environment would still generate the software for the rest of the buoy. Software to control the oil sensor would be hand generated, but would be produced to conform to the design of buoy sensor interfaces. That is, the same design and documentation standards would be applied to the oil sensor interface software as to the interface software for other buoy sensors. In this way, the same disciplines and approach used in developing standard buoy software may be applied to developing specialized buoy software.

For some systems, the application of Synthesis may be restricted to particular subsystems (CSCIs). This is likely when moving from a conventional development process to Synthesis, when large parts of a system are reused without change, or when it is not economical to apply Synthesis uniformly across a project. It may also occur when subcontractors are developing software using a conventional process. The preceding discussion of evolution then applies to the CSCIs where Synthesis is used. In such a case, the efficacy of the Synthesis process is strongly affected by how well the interfaces among CSCIs are defined. Where an interface is precisely defined, and changes to it can be forecast, it may be treated just as any other interface that must be analyzed as part of the domain. The application engineering environment for a CSCI where Synthesis is used must generate the code and documentation for parts that deal with the interface.

For example, the buoy system might be developed as an accessory to an existing shipboard command and control system. Buoys could be attached by cable to a ship and towed, transmitting their messages to the shipboard system via the cable. The entire buoy system might then be treated as a subsystem of the shipboard system, and the buoy software as a CSCI within it. Assumptions that the domain engineers make about the domain must include decisions about the interface to the shipboard system. The domain analysis for the buoy system can treat those assumptions the same as assumptions about the satellite navigation systems that the buoy uses.

The following presents details of the Synthesis process. Application engineering is presented first because an understanding of its activities and mechanisms help motivate the products and activities of domain engineering.

# 4. APPLICATION ENGINEERING

Synthesis frees the software developer from revisiting design and code for every new application and for new requirements for existing applications. During application engineering, the application engineer specifies an application model and generates deliverable products from it. As long as his requirements are within the bounds of the application domain, he can rapidly produce deliverable software from his model. Because production is rapid, the customer may frequently view successive refinements of the system and provide feedback to the application engineer. Figure 3 shows application engineering as part of the system development process.

The development process is cyclic. It starts with the user expressing his needs as a mission statement. The application engineer takes the mission statement as input, and uses the application engineering environment to formulate a precise technical statement of the customer's requirements. In concert with the environment, he rapidly produces a member of the application family, including deliverable documentation and code, that meets the requirements. The deliverables are given to the customer for validation; he may accept them or decide that further revision is needed and resume the cycle.

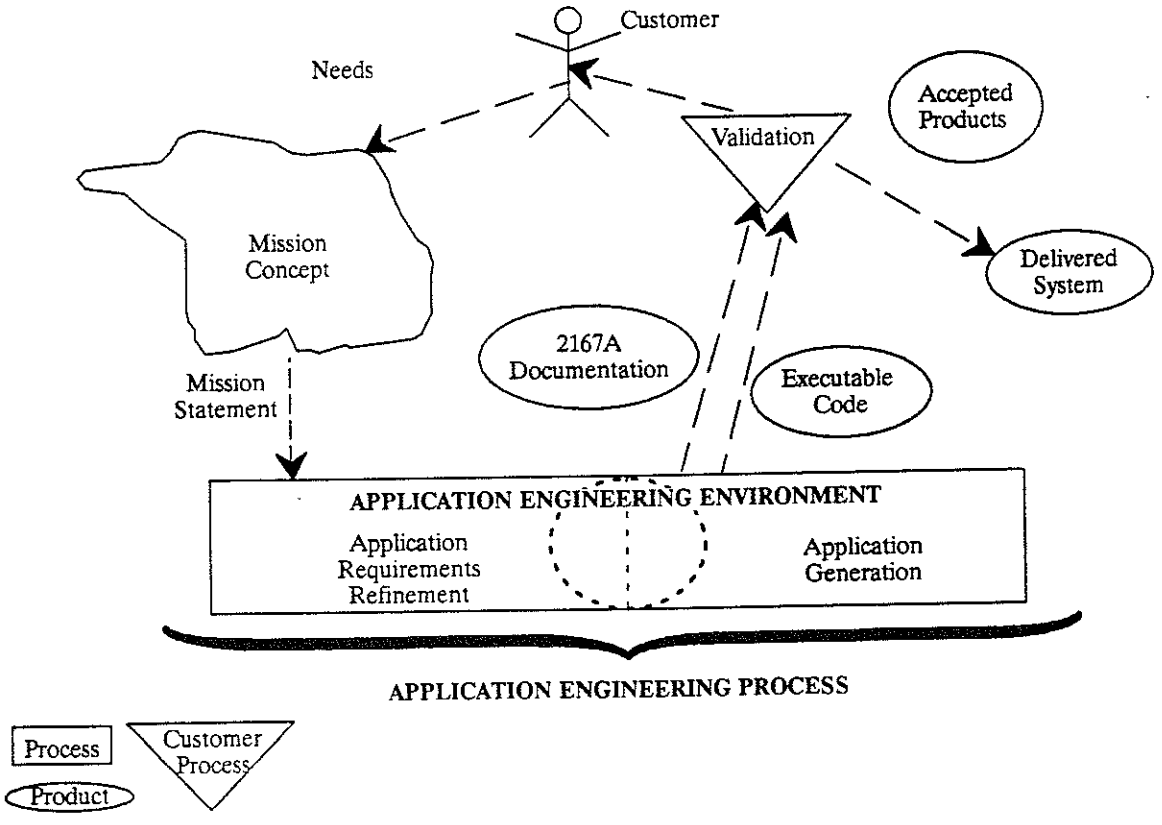



Figure 3. The System Development Process Using Application Engineering

## 4.1 THE APPLICATION ENGINEERING ENVIRONMENT

The application engineering environment helps both with specifying requirements for members of an application family and with mechanically generating deliverable products. Figure 4 shows the major interactions that occur between the application engineer and the environment.

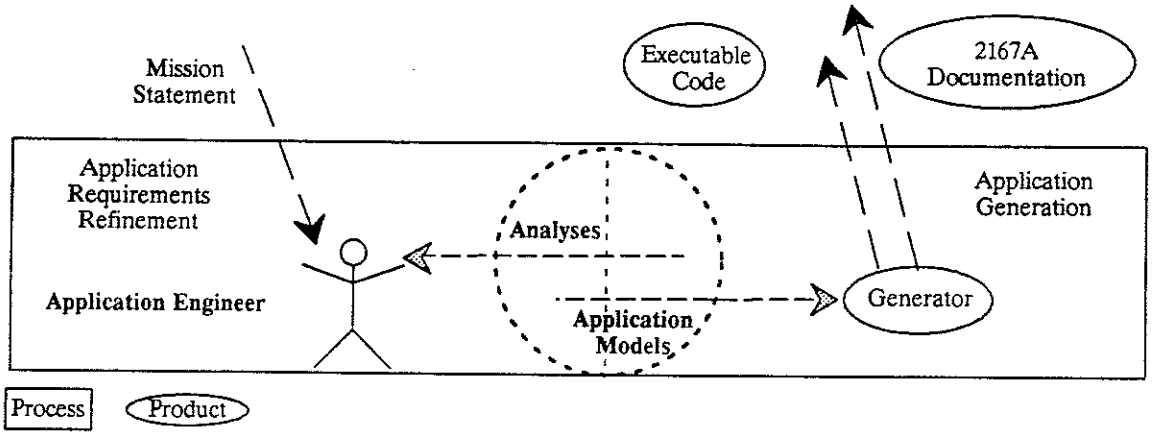



Figure 4. Interactions Between the Application Engineer and the Environment

## 4.2 APPLICATION ENGINEERING TASKS

The application engineer's tasks may be characterized as follows.

- Transforming the customer's input into a precise requirements specification for the member of the application family to be developed. This requirements specification is an application model. It is the only technical specification of the software that the application engineer produces.
- Analyzing the application model to ensure that it meets the customer's needs. Based on these analyses, the application engineer may refine the model many times before he is satisfied with it.
- Generating deliverable products, including documentation and code, for the application. Because the process is mechanical, little or no verification of the products is necessary. The refinement process and product generation process heavily emphasize validation rather than verification.

### 4.2.1 Transforming the Input

Mission statements tend to be general, prose descriptions of the customer's needs. For the HAS Buoy System (described in Section 3), they may include statements such as "The buoy shall be able to determine its location and report it to nearby vessels." Such a statement conveys a need that the system must satisfy, but is too imprecise to serve as a technical specification. For example, the accuracy of location determination, the precise characteristics of the communications channels to be used for reporting, the reporting frequency, and the definitions of "nearby" and "vessel" must all be precisely specified before a system design can be established. A primary task of the application engineer is to take such statements and, with the help of the application engineering environment, produce a technical specification for the application.

The application engineering environment provides him with a language, and associated editor, for representing application models. The language may be textual and/or visual. For example, he may select communications channels to be used on a buoy by choosing among visual icons or by writing a text statement. This language is the *application modeling language*, since it is used to build a model of the required system in application terms.

The engineer may use the application modeling language to:

- Select among the variations that distinguish among family members (i.e., he makes decisions that characterize the family member that he wants to generate and that differentiate it from other members);
- Modify an existing application model that is already close to what he needs; or
- Obtain information, including analytical results and characterizing assumptions, about the application domain. For example, he may request the results of a simulation of signalling on communications channels available for use within the application domain.

An application engineer can use an application modeling language to specify a member of the buoy family with four water temperature sensors, four wind speed sensors, and eight air temperature sensors. With each sensor, he specifies the rate at which it is to be sampled, the range of data to be collected, the resolution of the sensor, and the response time of the sensor. He might further add buoy-to-ship communications, an emergency switch and red light, Omega navigation, four receivers, eight transmitters, and two computers of 2.6 million instructions per second (MIPS) processing rate each. Finally, he might specify that the buoy broadcast a weather message on a four-second cycle.

### 4.2.2 Analyzing the Application Model

The application engineer must ensure that the application model satisfies his customer's requirements. The model must be complete, internally consistent, feasible to produce, and traceable to the mission statement. The environment provides a set of analyses that the application engineer may use to verify these properties. For example, if the engineer specifies that buoy weather reports are to be sent more often than the rate at which data can be sampled, the environment may so inform him. It would warn him when the processing rate for the computers is inadequate for the buoy configuration he has specified, and suggest that he increase it. It might also estimate the cost of producing the buoy. Where results are not available in closed form, the environment may provide simulations for the engineer to use. He may control such simulations in the same way that he specifies an application model; i.e., by choosing the type of simulation and the parameters to be varied during the simulation. The simulation may execute on the platform used to host the environment, or it may execute on the operational platform with code downloaded from the host.

### 4.2.3 Generating Deliverable Products

The application engineer may use the environment to generate code and documentation at any point in the application refinement process. He does this because he wants to run a simulation or other analysis based on the code, or because he believes that the application model satisfies the customer's requirements and is ready for customer validation. He generates the code mechanically. It is the deliverable source and executable code for the system. He also generates the deliverable documentation

mechanically. For DoD contracts, it conforms to DOD-STD-2167A. Figure 5 shows the elements of product generation.

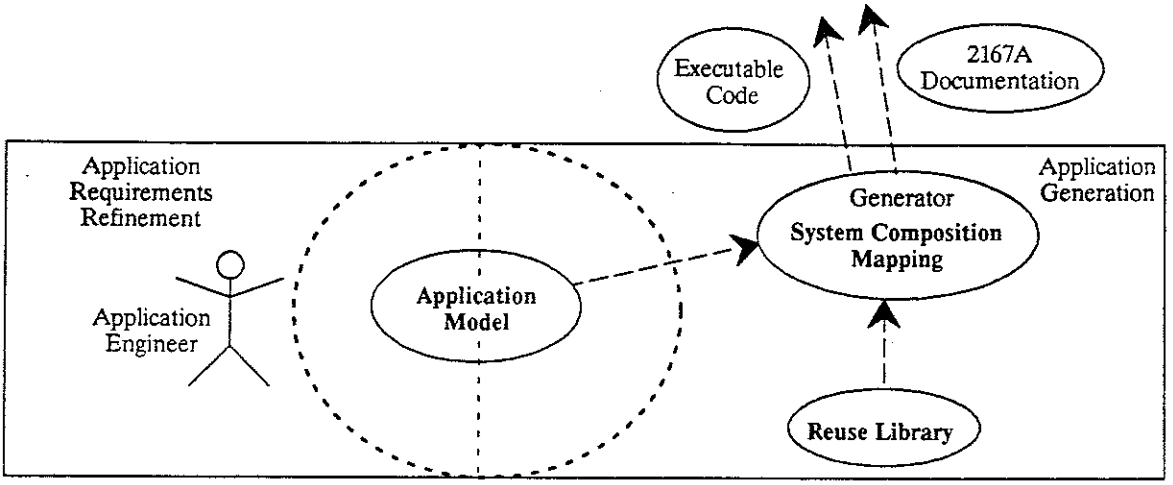



Figure 5. Product Generation Components of the Application Engineering Environment

## 4.3 PRODUCT GENERATION

To successfully generate code and documentation for members of an application family, the environment must contain a collection of components that can be mechanically adapted and composed to form different members of the family. Such a collection is called a *reuse library*. To take advantage of such a library, there must be a language (the application modeling language) in which the application engineer can specify the requirements (the application models) for the members of the family. The application models he creates must be mappable into specific members of the application family. Each model must mechanically induce the selection, adaptation, and composition of components drawn from the reuse library. These tasks are the responsibility of the *generator*. A critical part of the generator is the mapping from the application model to the reuse library to select components. This mapping is the *system composition mapping*. The three key components of product generation, then, are the reuse library, the application modeling language, and the generator, as shown in Figure 5.

### 4.3.1 The Application Modeling Language

The application modeling language satisfies two different concerns: specifying the requirements for an application, and embodying the decisions that guide adaptations of components.[1] The language allows the application engineer to concentrate on the requirements for his application without concern for how the code and documentation for the application is produced. The models expressed in the language represent variations on decisions that are common to the application family. For example, if the decision is that buoys must be able to determine their locations, then variations may be the sensors and algorithms used to determine location. The application language must then provide a means of expressing the particular sensors and algorithms to be used for a particular member of the family. The language is designed so that models can be mechanically mapped into a subset of the components in the reuse library, and so that models provide the data needed for adaptation.

1. The user interface and other external interfaces for the application are considered here to be part of its requirements.

### 4.3.2 The Reuse Library

The reuse library contains the components needed to generate code and documentation for members of the application family. Components in the library are designed to be adaptable to variations in the family. Each component is designed to represent a decision common to members of the application family. Different implementations of the component can be generated by a variety of mechanisms, such as instantiating parameters, completing templates, and conditionally including lines of code or documentation.

Variations described by an application model map directly to selections of particular components and variations on those components. For example, selection of a wind sensor with particular characteristics results in the selection and adaptation of the wind sensor component from the reuse library.

### 4.3.3 The Generator

The generation process should be mechanical. It can be implemented algorithmically, without heuristics. The key to mechanical generation is the system composition mapping. It relates elements of the application model to elements in the reuse library that provide the corresponding function. The application family must have a common software architecture that is adaptable across the domain. This common design is called a *reuse architecture*. The design is composed of adaptable components and the dependencies among them. The reuse architecture guides the stocking of parts in the reuse library.

The generator takes an application model and the reuse library as input. It uses the system composition mapping to identify the components of the reuse library corresponding to parts of the application model, as shown in Figure 5. It uses parameters specified by the model to instantiate a particular implementation of each component. It uses dependencies among the components to identify other components that must be instantiated. It also uses these dependencies to compose the instances to form a complete system, including documentation.

*This page intentionally left blank.*

# 5. DOMAIN ENGINEERING

Current manufacturing processes design products to be rapidly, easily, and efficiently manufactured. Automobiles, computers, bicycles, and other products are designed so that they can be customized to buyer specifications while still applying assembly line technology. Manufacturers achieve efficiency by elevating the importance of design-for-manufacturing in the overall product development process. Early in the process, they give significant forethought to the manufacturing process and how that process can be readily adapted to respond to market changes.

Similarly, domain engineering is the Synthesis process dedicated to designing for rapid, flexible production. The process institutionalizes the analysis of likely variations in the product over both its life span and the range of potential customers. For instance, in developing an avionics suite, domain engineers can consider likely changes to evolving requirements, such as supporting new electronic countermeasures, and likely variations among customers, such as the requirements of different NATO members. The result of domain engineering is a domain-specific software production environment, the application engineering environment. The environment is analogous to the assembly line for manufactured products. It generates applications in many variations rapidly, easily, and efficiently.

Domain engineering relies on two key concepts: (1) describing requirements for similar systems in similar ways, and (2) developing a common software architecture for a family of related systems. Just as automobiles can be described similarly, in terms of maximum speed, acceleration, fuel efficiency, interior room, and other factors, so can avionics systems, telecommunications systems, and others. In the example from Section 3, HAS buoys can be described in terms of the environmental parameters that they monitor, the frequency with which they send reports, and the types of requests to which they respond.

Just as there are common designs for automobiles, there are common architectures for different classes of digital systems. Such designs are the basis for efficient manufacturing. For example, each auto on an assembly line is built using the same basic design, with variations determined by the customer's requirements. Efficient manufacturing is possible because the product development process emphasizes forethought about the variations customers will desire. Both the cars and the manufacturing process are designed to easily accommodate these variations within the framework of the assembly line. A primary goal of domain engineering is to develop software architectures that result in the same property across a domain. The more similar the requirements of two applications, the smaller the difference in design between the two. Such architectures exist in domains as diverse as compilers, real-time executives, digital communications switches, power plant control systems, and spacecraft operations control centers. Taking advantage of them requires a systematic production process that can map variations in the description of an application into a variation on a design.

A major goal in the development of Synthesis is providing a set of methods for domain engineering that can be practiced as part of a well-defined and repeatable software engineering process in member

companies. This can be done by basing Synthesis on sound engineering principles such as information hiding, abstraction, and hierarchical structuring. Accordingly, domain engineering has two related objectives:

- Standardizing the way in which similar systems are described.
- Standardizing the products of application engineering so that the description of an application determines the unique content and form of those products.

Domain engineering consists of three major activities: *domain analysis*, *domain implementation*, and *domain management*. Domain analysis formalizes the expertise in a particular application area to create standard models for requirements representation and standard designs for the system architectures. The result is a specification for an application engineering environment. Domain implementation develops a library of reusable components and supporting mechanisms for producing code and documentation that satisfy such specifications. The result is an application engineering environment. Domain management allocates and monitors resources to do domain analysis and domain implementation. It is the process of managing a domain engineering project. Figure 6 shows how the products of domain engineering are used in creating an application engineering environment.

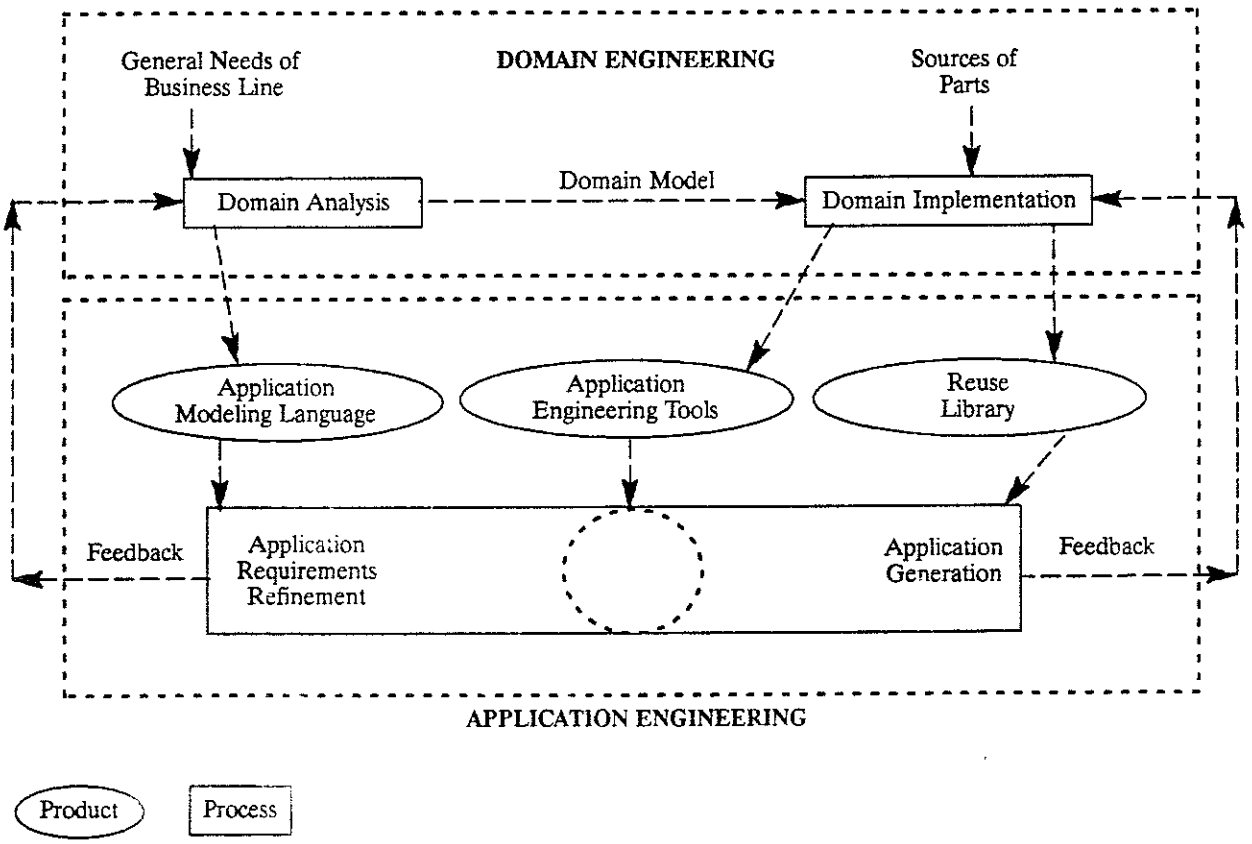



Figure 6. The Synthesis Process

## 5.1 THE DOMAIN LIFE CYCLE

The life cycle of a domain begins with identifying a business area need. It evolves as the understanding of this need grows or changes, and it ends when the needs of the business area are no longer consistent

with the original concepts of the domain. The life of a domain may coincide with the life cycle of a single complex system, or it may encompass the lives of many similar but differing systems.

The common issue is requirements variation. Whether building a single, long-lived system for a single customer or many distinct systems for many customers, a primary obstacle to productivity is misunderstood or changing requirements. Building to fixed, unambiguous requirements is relatively easy. Unfortunately, most useful systems are characterized by poorly understood and changing requirements. It is often possible to solve this problem by constructing a unified, adaptable solution that accommodates those variations in the context of a common design. This is done by explicitly studying and accommodating the scope of expected variations in the requirements, whether for a system or set of similar systems. All variations cannot be anticipated, but experience in an application area is sufficient to anticipate many of them, and categorize and abstract from many more. The scope of a domain is the set of applications that can be accommodated by varying the set of requirements and engineering decisions that distinguish domain members.

The scope of the domain must evolve as the needs of a business area change. The applications characterized by a domain are fixed only in the broadest terms. While needs may disappear, the usual trend is toward new and expanding needs.

## 5.2 DOMAIN ANALYSIS

If differences in software solutions can be derived from differences in the problem requirements, systems having similar descriptions will have similar solutions. Furthermore, a particular change in a description will have predictable effects on the solution. Domain analysis determines how systems should be described so that this problem/solution correspondence holds. Establishing the correspondence allows application engineers to think of a system totally in terms of its application model.

The product of domain analysis is called a *domain model*. A domain model has three parts:

- A conceptual framework for application models: The syntax and semantics of a language for describing application models appropriate to the domain.
- A reuse architecture: A common system architecture that is used to structure a reuse library and its contents as adaptable components, including both documentation and code.
- Product composition mappings: Proceduralized specification of the relation between the application modeling language (characterizing possible variations in requirements) and components of the reuse architecture (characterizing available solutions). These mappings determine how life-cycle products are derived using adapted, reusable components.

For the HAS buoy system, the conceptual framework consists of the terminology and other technical models used to specify the behavior of buoys (sensor types, communications links, possible messages) and relations among those concepts. A reuse architecture for HAS buoys consists of a set of objects that permits changes to be made independently to sensor characteristics, message types, communications links, and rates of gathering, filtering, and transmitting data or messages. A standard architecture for a sensor fusion system can be used. The product composition mapping associates the objects defined by the architecture to those needed to implement a particular application model; e.g., linking objects to implement a particular sensor type, message type, etc. It also associates the documentation

containing their specifications to elements of the application modeling language used to describe sensor characteristics, required messages, and so on.

## 5.3 DOMAIN IMPLEMENTATION

Domain implementation provides automated support for domain-specific application engineering. The products of domain implementation are:

- A reuse library, implementing the reuse architecture by implementing adaptable system components. These may be newly created or reused from existing systems.
- An application engineering environment for constructing, evaluating, and generating required products of an application model.

The domain model (i.e., the modeling language, reuse architecture, and composition mapping) serves as the specification for the products of domain implementation.

There are many opportunities for reuse in building application engineering environments. Application engineering environments can also be treated as domains. The principles applied to other software products can also be applied to the software required to implement an application engineering environment. Thus, the productivity benefits of Synthesis can be realized in producing the software products that support the process itself.

## 5.4 DOMAIN MANAGEMENT

Domain management determines the best use of available domain and software expertise to support the needs of application engineering projects most effectively. Its functions are the same as those of conventional project management applied to the domain engineering activities. Priorities for domain evolution derive from the needs of those projects and the anticipated needs of future business. To be considered successful, domain engineering must be responsive to the schedule pressures of application engineering projects.

# 6. WHY A SYNTHESIS METHODOLOGY IS FEASIBLE

Although experience with Synthesis is limited, there are sound reasons for believing that it is a credible approach to the software productivity and quality objectives of member companies. First, the approach to developing a practicable Synthesis process is based on existing technology. While the overall process differs dramatically from conventional development, each of the component pieces can be effectively applied to develop embedded system software. In addition, there is a growing body of evidence that a disciplined development process based on systematic reuse can yield significant productivity gains. While technical issues remain, and the viability of the overall process has yet to be demonstrated in a member company environment, results to date are encouraging enough to support continued effort both by the Consortium and other applied research groups. For instance, the Software Engineering Institute is investigating a similar approach in their Domain Specific Software Architecture (DSSA) work, and the Defense Advanced Research Projects Agency (DARPA) is funding work in related technologies (under the heading "Megaprogramming").

## 6.1 RELATED TECHNOLOGIES

Synthesis combines aspects of rapid prototyping, application generation technology, and object-oriented design. To understand why the Synthesis process takes the form it does, it is worth briefly comparing it with these technologies.

In the requirements analysis stage, application engineering under Synthesis looks much the same as it does under a rapid prototyping model. It starts with an inexact description of requirements (e.g., a mission statement) and evolves to an exact description through successive refinement of executable models. The difference is that, for rapid prototyping, the result is a specification that still must be realized through design, coding, and testing. Under Synthesis, the products of each successive refinement are in a production-quality form, so that the next step is customer-oriented validation.

Application engineering under Synthesis has the same goal as "automatic programming." It potentially automates the generation of code and other products from requirements specifications. It differs from most automatic programming approaches because it is based on the adaptation and reuse of components rather than transformation of one abstract representation into another. Furthermore, there is no attempt to achieve generality. Synthesis is geared to developing families of similar systems. While more limited in potential power than transformational schemes, a reuse-based approach has the advantage of being realizable with current techniques.

Synthesis takes an object-oriented approach to the design of reusable software components. Like all object-oriented methods, this approach uses objects to abstract from unnecessary implementation detail and encapsulate information likely to change. These principles form a basis for developing

software components suitable for reuse. Objects, object classes, and class hierarchies provide the means to represent design decisions common to applications of a family and to make adaptable those details that vary. The approach extends the usual object-oriented model in two ways: it provides practical guidance on how to design a system of objects so that it is structured for reuse, and it provides additional ways to represent abstract classes that generalize collections of similar objects.

## 6.2 KEY ELEMENTS OF FEASIBILITY

Previous sections have discussed the component parts of Synthesis and their roles in the overall process. This section describes the supporting technologies available and notes where these technologies have been applied with some success. The intent is not to validate the process, since this would require complete development of a real system (a goal of the Synthesis Project). It is to show that candidate technologies are currently available and that these technologies can satisfy the overall aims of Synthesis. The evolution of these technologies and acquisition of additional Synthesis technologies is another goal of the Synthesis Project.

### 6.2.1 Components of Domain Engineering

#### 6.2.1.1 Domain Analysis

**Application Modeling Language.** In defining an application modeling language, the goal is to design a language that is precise enough to permit a complete description of requirements in terms of application domain concepts, and yet provide an understandable representation of the family of systems (i.e., what can vary) that is producible from the reuse architecture.

Application generators and other examples using application modeling languages make it clear that *ad hoc* approaches can work (Blum 1988; Feiler 1986; Lee 1989), even for real-time embedded applications like flight control (Coron 1988). Previous work in developing models to describe embedded software requirements (Heninger et al. 1978; Faulk and Clements 1987; van Schouwen 1990) shows that a specification language can be expressed in application-oriented terms and yet be sufficiently precise to use for a complete technical specification. These techniques have been extended to create the application modeling language for the Spectrum domain-independent application engineering environment prototype. Attempts to define the characteristics of a good software specification (Balzer and Goldman; Winograd 1979) show that the techniques can be applied as part of a complete Synthesis process and that the approach to modeling languages is generalizable.

While a repeatable method for developing application modeling languages is still in the process of being defined, a prototype process has been successfully applied in a domain of CASE design tools (Burkhard et al. 1990). Modeling language issues are currently being investigated, such as how best to represent variations among family members. Subsequent applications of the prototype process will extend the current results to member company domains.

**Reuse Architecture.** In developing system components for reuse, the concern is both with the overall software architecture and with the design of individual components. Since the system architecture determines the dependencies among design artifacts, it determines which adaptations are easy to make and which require redesign. The goal of developing a reuse architecture under Synthesis is to capture the allowed variations among family members at the level easiest to change—the detailed design objects. Representing variations is the job of the designers of the reusable subprograms (and their documentation). This requires methods for specifying the design of adaptable components.

The Synthesis process takes an object-oriented approach to design, with a decomposition strategy aimed at reuse. The design approach is based in part on information hiding (Parnas 1972) and the notion of developing systems as members of program families (Lamb 1988; Parnas 1976; Parnas 1978). Experience in developing the components of Spectrum (Campbell 1989) and the Design Composer Domain shows that these design techniques can create standard, reusable software architectures using an analysis of expected variation (Burkhard et al. 1990; Jaworski et al. 1990). Earlier work at Bell Northern Telcom (Lasker 1979) and Bell Laboratories (Hester, Parnas, and Utter 1981) in using these techniques to develop maintainable, reusable architectures also lends credibility to the approach. Current investigations include extending the approach using object-oriented analysis techniques that incorporate the benefits of data-flow approaches and graphical specification (Ward 1989).

The recent growth in object-oriented approaches has resulted in a number of design methods aimed at designing and specifying adaptable components, usually in the form of abstract data types. Examples include (Meyer 1988; Booch 1987; Ward 1989; Lamb 1988; Clements et al. 1984). These methods must now be incorporated to make a complete Synthesis process, and the entire process must be applied to a member company domain. The Consortium is currently creating standards for the design and representation of adaptable components, based on these works, with a focus on translation to Ada.

**Composition Mapping**. A composition mapping formalizes the relationship between variation in system requirements (as represented in the application modeling language) and corresponding variability in the design (as embodied in the reuse architecture). The composition mapping specifies exactly what should change about the design in response to a particular variation in a given requirement.

Experience in creating composition mappings for Spectrum and the Design Composer Domain suggest that the process depends on both the domain and the extent of variation represented. Minor variations or mappings restricted to allocating subsets of components (e.g., Lasker 1979) are not technically difficult to represent. At issue are the processes and techniques for complex mappings that preserve time-critical performance characteristics of real-time software. However, related technologies (e.g., in optimizing compilers) may be applicable, and other work offers a variety of approaches that are effective for some types of applications. These include compiler-like techniques in VAPS (Virtual Prototypes, Inc.), refinement rules (Smith 1990), and straightforward table-lookup (Coron 1988), in addition to the procedural forms the authors have used.

#### 6.2.1.2 Domain Implementation

**Reuse Library**. Much of the recent work in constructing reuse libraries focuses on searching a library for components or their aggregations to satisfy a particular set of characteristics (Frakes and Gandel 1989). The reuse strategy supported by these approaches can be described as *opportunistic reuse*. The library serves as a common resource across projects. Library parts are used, as opportunity permits, to satisfy a particular set of requirements. The library organization and search strategy attempt to make the best of reuse opportunities across a broad class of applications.

In contrast, the Synthesis approach can be described as *systematic reuse*. Not only are individual components reused, but the organization of the system itself (both the parts and relations) is reused. The applicability of the library narrows to a particular domain that uses a common system architecture. While the general applicability of an opportunistic strategy is lost, the problems of organizing and searching reduce to the problems of constructing an appropriate reuse architecture and composition mapping. The

library should not be searched based on arbitrary requirements, but only over the range of variation covered by the domain. Abstraction-based reuse (Campbell 1989), a specific form of systematic reuse, captures variations among similar components as parameters using a metaprogramming representation. This allows the tracing from requirements to corresponding components to be largely "hard-wired."

The structure of the reuse library reflects the structure of the reuse architecture (i.e., the object decomposition structure). This follows naturally since the decomposition structure is created based on anticipated variation. The approach has been effective in developing the reuse library for Spectrum and the Design Composer Domain. Other organizations with the same property of a fixed-structure reuse strategy have also been effective in other application areas. For instance, the reuse repository for GALA (Coron 1988) has a structure based on functional decomposition (i.e., flight control functional components such as filters and gates). TEDIUM (Blum 1988) uses a structure based on a classification taxonomy for information system components (similar to the Booch taxonomy).

**Application Engineering Tools.** An application engineering environment capable of generating products requires populating the reuse library with the required components (code objects, documentation, and test procedures) that are adaptable to anticipated variations, and creating a generation mechanism. A fully automated environment also requires facilities for the input, viewing, analysis, and storage of application models. While full automation is not essential, it makes application engineering much faster and more effective. Experience building Spectrum (Campbell 1988) shows this to be a feasible task, but requiring significant effort. The reuse methods discussed are important in making it feasible. Few of the required capabilities are unique to application engineering environments. Thus, technology developed for other systems that store, manipulate, and present organized systems of design information (e.g., CAD/CAM) can be exploited.

### 6.2.2 Components of Application Engineering

**Application Specification.** Creating an application specification in terms of an application modeling language is analogous to rapid prototyping. Instances of successful application modeling can be found for most of the tools cited (e.g., Spectrum, AutoCode, GALA, and TEDIUM). An open process issue concerns how to best integrate specification activities in an overall description of an application engineering method. In concert with in-house experiments in applying the Synthesis process, work is underway to develop practical guidance for future Synthesis practitioners.

**Validation and Assessment.** The problems of validation and assessment are not different under Synthesis than under any other approach to software development. An application model is no more than a data representation of requirements and engineering decisions that characterize a system within a domain. Technologies for validation and assessment of such models are immature but needed equally under any methodology. The Consortium's Dynamics Assessment Project is successfully developing methods for the modeling and performance assessment of some systems (Software Productivity Consortium 1989a; Software Productivity Consortium 1989b). The initial goal in this area is to encompass those methods within the Synthesis methodology.

**Product Generation.** The current Synthesis approach focuses on conventional and well-understood technology for instantiating adaptable components (Campbell 1989). This approach includes capabilities similar to but beyond the use of instantiation parameters (e.g., as are used to create instances from Ada generics), macros, and code templates. These technologies for reuse-based generation have been used in Spectrum, the Booch parts (Booch 1987), CAMP (McNicholl, Palmer, and

Mason 1988; Palmer 1989), and GALA (Coron 1989). Experiments with Synthesis have created parts and generated systems in the CASE design tool domain (Burkhard et al. 1990). Subsequent experiments will focus on member company domains.

There are a number of other generation technologies based on transformation strategies (as opposed to reuse) being applied (e.g., Blum 1988; Smith 1990). These technologies can be used in conjunction with reuse technologies and will be explored as that work matures.

## 6.3 EXPERIENCE IN PRACTICE

The commercial sector appears to have applied only the application generation part of the Synthesis process. From commercial sources, instances of narrow, domain-specific application engineering environments have started to appear and gain adherents. Examples include GALA (Coron 1988), ISI AutoCode ($MATRIX_x$) (Integrated Systems, Inc. 1988), and Foresight (Athena Systems, Inc.). Significant productivity improvements have been reported where these environments fit the needs of a project. They differ from the Synthesis approach in that domain engineering is transparent to their clients. They do not include the end user in the development and modification cycle of the application engineering environment itself. Under Synthesis, the construction and evolution of the environment is an integral component of the software development process.

Development and use of Spectrum (AI Research Highlights 1986; Campbell 1988; Software Architecture & Engineering, Inc. 1986) has shown the ease with which software can be built when its requirements are expressible as an application model. Spectrum is based on the use of a Synthesis-like reuse architecture and system composition mapping. Its application modeling interface is domain-independent but representative of the approach for domain-specific application modeling. Its construction and the construction of derivable applications are based on the software engineering methods discussed, including information hiding, modularization, abstract interfaces, and families of reusable components.

The complete Synthesis process (in prototype form) has been experimentally applied at the Consortium to the domain of CASE design tools (Burkhard et al. 1990). The Design Composer Domain task applied the Synthesis process to develop an application engineering capability for generating CASE design tools. The process resulted in a complete set of Synthesis work products and generated two versions of design tools supporting different methods (the Consortium method and Structured Design). The experiment followed a complete Synthesis process as closely as possible, except that no environment for automated application engineering was constructed. The project also leveraged existing reusable components and associated generational technology of Spectrum so that not all new components had to be created. This demonstrates that the general method and work products are correct. What remains to be demonstrated is that these methods will be equally effective in a member company domain. The current work in the Consortium's Validation Laboratory proposes to do this.

STAGE[2] (Craig 1988) and Bofors (Funk 1990) are other examples where a a company has taken a domain-specific approach to organizing application development work. STAGE is used at AT&T Bell Laboratories to build embedded microprocessor software for telecommunications products. It takes a very similar approach to that of Spectrum, but emphasizes domain analysis for producing a

2. STAGE is now known as MetaTool.

domain-specific language. For distributed real-time embedded systems, the Bofors work supports many of the principles advocated for Synthesis. In particular they report success with reuse, iterative development methods, integration of system engineering and software architecture, and incremental integration of system "threads."

## 6.4 ECONOMIC ANALYSIS

The value of Synthesis to improve productivity and quality of member company software development can only be proven through realistic use. However, a preliminary analysis of its economic characteristics supports the thesis that Synthesis reduces the typical cost factors of a software systems life cycle. These factors include:

- Development costs, from mission statement to initial product delivery.
- Software and system integration costs.
- Maintenance costs to accommodate changed customer requirements and to correct errors.
- Resource shortages (schedule, budget, and personnel).
- Quality shortcomings (reliability and performance).

How these factors are approached in practice indicates how Synthesis can realistically improve productivity.

In developing software, experienced engineers routinely but informally adapt and reuse previous solutions. A problem seldomly requires a completely original solution. When the engineer encounters a familiar problem, he starts by examining the similarities to past problems and their solutions. From these, he creates a similar solution suited to the particular facets of the new problem. Synthesis aids this practice by identifying problem families and associated adaptable solutions. A significant component of development costs is the effort for refining requirements until they accurately reflect actual customer needs. Synthesis reduces this effort by providing for a domain-specific requirements language from which tailored software components (code and documentation) are created automatically.

Many of the problems in the development of a system do not show up until integration is attempted. These problems are traceable to ambiguities and errors in requirements and to communication failures among developers. The use of a standard requirements language and automated product generation means that such problems will arise much earlier in the requirements refinement process. Problems that trace back to domain engineering can be resolved uniformly for the domain as a whole.

In current practice, maintenance requires retracing the steps of initial development to understand the implications of a change. In going from requirements through design to an implementation, many assumptions and interacting decisions are made. The assumptions, decisions, and the effects of these decisions on the resulting code are seldom documented well enough to provide a reliable basis for maintenance. Instead, the maintainer must understand how changes in requirements affect assumptions and decisions leading to code and documentation changes. Synthesis concentrates this expertise into domain engineering so that both development and maintenance become an identical process of requirements analysis and refinement. The application engineering environment consistently transforms the application model into executable software and associated documentation products.

Domain engineers address resource shortages by creating standardized, adaptable solutions. These engineers have the knowledge and experience to recognize the fundamental concepts that characterize a domain and the issues that determine how different systems should be built. The ideal to which domain engineers work is that similar problems have similar solutions. By making key decisions that determine how a particular problem should be solved, much of the conventional effort of software development can be concentrated in the shared work of a domain engineering group.

Quality shortcomings are inevitable when software components are immature. Although, in principle, developer expertise is based on previous experience, specific component designs and implementations are usually redeveloped rather than reused. Because Synthesis makes reuse easier than redevelopment, the use of proven components dominates new development. Systematic reuse produces significant quality improvements over time. The domain engineering process aids this process by verifying reusable components and their anticipated adaptations before actual application.

Figure 7 is a top-level economic model of the cost factors of Synthesis. The conjectures, based only on limited anecdotal evidence, provide predictions for the economic benefits of Synthesis. Conjecture 1 predicts that the conventional cost of a single software development is a lower bound on the cost of domain engineering over the domain life cycle. A domain consisting of exactly one stable, well-understood application would be most likely to reach the lower bound. The upper bound is a function of the cohesiveness of the domain. It assumes reasonable diversity among a set of similar applications. Conjecture 2 predicts that the cost of an application engineering life cycle (excluding its attributable domain engineering support costs) will be significantly less than the conventional life-cycle cost of an application. The actual cost for a particular application will depend on the maturity of the domain and closeness of fit of the application to the scope of the domain. The implication of these conjectures is that the full life-cycle cost of a domain should be in the range of one times the cost of a single system development to two times the full life-cycle costs of a single system. Data will be collected in validation and pilot projects to evaluate the accuracy of these predictions.

Cost effective use of Synthesis requires recognizing that a domain must change as business needs change. Domain engineering incrementally captures and evolves corporate expertise in a business area. A domain can comprise a single, large, long-lived system that undergoes significant change, or it may comprise a large family of similar systems. In either case, the domain model and resulting reuse library and environment emphasize the most immediate concerns of the next required delivery of the large system or the next system of the family to be delivered. Domain engineering does not have to be perfect or complete before it is used in application projects. The needs of such projects provide the best guidance for how a domain should evolve. The value of domain engineering comes in systematically anticipating and planning for likely variations in future project needs.

A concern for member companies is the degree to which existing investments in software technology can be leveraged under Synthesis. A major advantage is that extensions of conventional technology and development practices are at the core of the current approach. Reengineering of existing software for adaptive reuse is practical and treated as a major activity within domain implementation.

The crucial decision is the long-term organizational commitment of key resources to a domain rather than to individual projects. This commitment involves not only software engineering expertise, but also comprehensive domain problem and solution expertise and the shared application of that expertise to support projects. The costs associated with this commitment will produce significant savings when applied over multiple projects, project iterations, or requirements changes.

| | |
|---|---|
| $N_A$ | number of applications expected to be delivered |
| $N_{AV}$ | expected number of versions/releases to customers of each application |
| $N_{AI}$ | expected number of iterations to produce an acceptable version/release |
| $C_{DN}$ | cost of domain engineering effort not directly attributable to a project |
| $C_{DI}$ | cost of direct domain engineering support for an application engineering iteration |
| $C_{AI}$ | cost of each application engineering iteration |
| $C_C$ | full life cycle cost for conventional application: $C_{CD} + C_{CM}$ |
| $C_{CD}$ | conventional development costs for an application |
| $C_{CM}$ | conventional maintenance costs for an application |
| $C_F$ | full domain life cycle cost: $C_{DN} + N_A * (C_{AI} + C_{DI}) * (N_{AV} * N_{AI})$ |

conjecture 1: $C_{CD} < C_{DN} + N_A * C_{DI} * (N_{AV} * N_{AI}) < 2 * C_C$

conjecture 2: $C_C / 100 < C_{AI} * (N_{AV} * N_{AI}) < C_C / 10$

implication: $C_{CD} + (N_A * C_C / 100) < C_F < (2 * C_C) + (N_A * C_C / 10)$

Figure 7. A Domain Life Cycle Cost Model

# 7. ISSUES AND OBSTACLES

The Synthesis experience has been primarily empirical in nature and, as such, offers evidence for, but does not prove, viability. To make Synthesis viable for routine practice, three broad areas of concern must be addressed. These are:

- Needed advances in supporting methods and technology.
- Strategies for incremental transition from current practices.
- Customer acceptance.

These concerns are in order of increasing difficulty, in terms of the Consortium's ability to ensure success.

## 7.1 NEEDED ADVANCES IN METHODS AND TECHNOLOGY

The Synthesis process is reasonably well-defined. The advances needed to make Synthesis practical are discussed in terms of domain engineering and application engineering.

Domain engineering is weakest in defining systematic methods for identifying stabilities and variations throughout a domain. Practical guidance for doing so needs further development. Such methods must be particularly sensitive to supporting incremental analysis and implementation of a domain.

A second concern in domain engineering is in methods for designing application modeling languages. The prime concern of the language is to permit the application engineer to express his requirements. The domain analysis must also provide for mapping from application models expressed in the language to the reuse architecture so that those models can be implemented using the product generation mechanism. The differing concerns of these two issues makes definition of a good language a difficult task. A systematic approach is needed.

Other concerns in domain engineering include the notation(s) to specify abstract components to be placed in the reuse library and to generate documents. In both cases, techniques exist but have been used only on a relatively small scale. Supporting technology is limited, and the technology for building automated environments is immature.

The primary concern for application engineering is how to structure and manage the process. Because there is little experience in applying Synthesis, and because application engineering represents a radically different approach to developing applications, the necessary process management techniques are not known. Developing an effective synergy between domain management and project management is a significant concern.

As noted earlier, technologies for supporting validation and assessment are immature. Much better support is needed to make the construction of large and resource sensitive systems viable under Synthesis.

## 7.2 STRATEGIES FOR TRANSITION

A gradual transition to Synthesis is a much greater obstacle to viability than advances in methods and technology. Introducing a complete Synthesis practice in one step, where there are no prior constraints, would be a relatively easy undertaking. The realities of an ongoing business and of employee acceptance make this unlikely.

The preferred strategy is incremental. Current practice would be adapted by degrees. There are several candidates for such adaptation:

- Using domain analysis to better understand and communicate the requirements of similar systems. Attempts to model a system in precise terms produces immediate benefits in raising issues and resolving ambiguities before costs of design, implementation, or testing are incurred.

- Introducing a process of reengineering of existing software components to make them adaptable to new uses (i.e., reusable). Past attempts at reuse have been limited by the inflexibility of software built implicitly for a single use. Adaptation-oriented reuse (of which object-oriented reuse is a restricted instance) makes adaptation decisions an explicit part of the construction of components, allowing the developer to anticipate many uses.

- Introducing an application engineering environment as a tool for system prototyping, either in support of contract procurement or as a software requirements risk reduction measure in the systems engineering process. This adaptation avoids much of the risks and costs of domain engineering, but leaves significant productivity and quality gains unrealized. Such an adaptation is feasible if an environment is already available in an adequate form. However, few such environments exist, either with a domain-independent orientation or for certain very narrow domains.

A second strategy is progressive commitment. Companies may initially commit a few low-risk projects to using Synthesis, perhaps by creating a technology center for the application of Synthesis. As the center gains experience on realistic projects and is able to show benefits, more projects can be committed to the use of Synthesis.

A key component of transition is the creation of technology transfer and training materials to help engineers become fluent in the practice of Synthesis. This transition will be most effective by providing commercial-quality courses and documentation.

## 7.3 CUSTOMER ACCEPTANCE

Customer acceptance is by far the greatest challenge to the viability of a new methodology. Despite the problems of current practice that Synthesis might solve, any uncertainty about the effects of its use requires some degree of risk-taking by the customer. Ultimately, the only reasonable solution is via a process of progressive commitment. By this process, early applications are in areas where risk

of failure is low and potential for payoff is high. Greater risk is accepted only when the payoff has been shown to be real.

In the case of Synthesis, there are two major issues. The first is how open customers of member companies are to a new process. If a customer requires the appearance of a conventional process, then there will be additional effort to produce and review the products associated with that process. This will not cancel the benefits of Synthesis, but it will reduce the benefits. The second major issue is whether customers are willing to contribute or reimburse costs associated with domain engineering. To succeed, current practices that discourage or prohibit use of previously built components must change for any reuse-based approach.

*This page intentionally left blank.*

# 8. CONCLUSION

Evidence for the viability of a Synthesis methodology is growing, not just at the Consortium, but in other industry, government, and academic efforts. Synthesis is well suited to the needs and environments of members companies, where expertise for building needed systems is scarce, multiple instances of similar systems are often developed, and applications have long lives and undergo significant change. The realities of software development are much closer to the ideal of Synthesis than to the traditional waterfall model. The primary obstacles to success derive from long-standing organizational and practitioner orientation to the development of hand-crafted, one-of-a-kind systems, and from procurement practices that discourage reuse. A major goal of the Consortium is to show that Synthesis has sufficient power to overcome these obstacles.

Changes in business practices can be costly. However, the Japanese have shown us that failure to produce the highest quality products, in response to customer needs, at the lowest possible cost, results in the loss of markets and ultimately in an inability to compete on a global scale. Software development will succumb to the same risk if the opportunity to apply engineering discipline for a sound, managed process is not realized. Current manufacturing trends are to redesign products and manufacturing processes to achieve both product goals (such as function, performance, or cost) and process goals (such as ease of manufacture, maintenance, or manufacturing flexibility) on the basis of continuing analyses of process characteristics. The philosophy behind these changes is that life-cycle costs and broader business needs dominate the issues of producing a single product. Market pressures do not allow for processes that fail to ensure quality, are costly to manufacture or repair or do not meet customer needs. Synthesis is an attempt to apply similar principles to software production in a way that meets the needs of member companies and their customers.

A Synthesis methodology can become a viable means to achieve member company objectives of high productivity of quality software. To realize this potential, active member company participation is critical. The growth and transition of a methodology into practice depends on application expertise and realistic use. This can only be realized with member company participation in pilot projects. Ultimate acceptance by projects depends on advocacy by member company corporate and line management, as well as technologists. The immediate challenge is to gain the support of these people for Synthesis pilot projects. Such projects are the best vehicle to advance the understanding of Synthesis and demonstrate its value.

*This page intentionally left blank.*

# GLOSSARY

| | |
|---|---|
| Application engineering | A process for rapidly creating production quality software (including documentation and code) from reusable components. The process constructs an application model to represent system requirements and engineering decisions. |
| Application engineering environment | The generator, reuse library, and any other components (such as analysis tools) provided to support the process of application engineering. |
| Application model | A specification of a particular system's requirements in the form of an application modeling language. It is a product of application engineering. A complete model provides a sufficiently detailed specification of system requirements and systems engineering decisions to mechanically generate code and documentation from reusable components in the reuse library. |
| Application modeling language | A product of domain analysis. The application modeling language provides both syntax and semantics for describing system requirements and systems engineering decisions for applications in a domain (in terms of an application model). |
| Domain | Implementation of system requirements (in terms of an application model) is based on a mechanical process of adapting and composing reusable components. The family of applications that can be generated from these components and composition rules is a domain. Conversely, the term is used to encompass the set of systems for which one is building such an application engineering environment. |
| Domain analysis | Formalizes the expertise in a particular application area to create standard models for requirements representation and standard designs for system architectures. The results is a domain model. The domain model serves as a specification for the subsequent domain implementation process. |

| | |
|---|---|
| Domain engineering | A process for creating an application engineering environment for a family of similar systems. Domain engineering includes all of the activities associated with identifying a target family of applications, capturing their variation, constructing an adaptable design, and defining the mechanisms for translating requirements into systems created from reusable components. Domain engineering consists of domain analysis and domain implementation |
| Domain implementation | Develops an application engineering environment. This includes a library of reusable components and supporting mechanisms for producing code and documentation that satisfy specifications written in terms of the corresponding application modeling language. |
| Domain management | Allocates and monitors resources to do domain analysis and domain implementation. It is the process of managing a domain engineering project. |
| Domain model | The product of domain analysis. The model consists of an application modeling language, a reuse architecture, and their system composition mapping. |
| Generator | Mechanizes product creation for application engineering. It is constructed from the specifications of the reuse architecture, referencing the application modeling language and the system composition mapping. The generator implements the mapping by tying together elements of the modeling language and elements of the reuse architecture so that code and other products can be generated from a particular model. |
| Reuse architecture | The common software architecture of an application family that is adaptable across the domain. The design is composed of adaptable components and the dependencies among them. The reuse architecture guides the stocking of parts in the reuse library. It is a product of domain analysis. |
| Reuse library | A collection of components that can be mechanically adapted and composed to form different members of a family of applications. It contains the parts needed to generate code and documentation. The reuse library is a product of domain implementation. |
| System composition mapping | Relates elements of the application model to elements in the reuse library that provide the corresponding function. It is a product of domain analysis. |

# REFERENCES


AI Research Highlights 1986 — Software A&E Readies Spectrum Software Environment. *AI Research Highlights*. New Science Associates, Inc. 136-139.

Athena Systems, Inc. — 139 Keifer Court, Sunnyvale, CA 94086

Balzer, Robert, and Neil Goldman — *Principles of Good Software Specification and their Implications for Specification Languages*. USC/Information Sciences Institute.

Blum, Bruce I., 1988 — An Illustration of the Integrated Analysis, Design and Construction of an Information System with TEDIUM: Application Specification and Documentation. In *Computerized Assistance During the Information Systems Life Cycle*. Edited by L. Bhabuta. Elsevier Science Publishers.

Booch, Grady 1987 — *Software Components with Ada Structures, Tools, and Subsystems.* Menlo Park, California: Benjamin/Cummings.

Brooks, Frederick P., Jr. 1987 — No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*.

Burkhard, Neil, Jeff Facemire, James Kirby, Jr., and Jim O'Connor 1990 — Applying Synthesis in the Design Composer Domain (Draft). Herndon, Virginia: Software Productivity Consortium.

Campbell, Grady H., Jr. 1988 — Abstraction-Based Environments (Draft, unpublished). Software Architecture & Engineering, Inc.

1989 — *Abstraction-Based Reuse Repositories*. Herndon, Virginia: Software Productivity Consortium.

Cleaveland, J. Craig 1988 — Building Application Generators. *IEEE Software*. 25-33.

Clements, P. C., R. A. Parker, D. L. Parnas, and J. Shore 1984 — *A Standard Organization for Specifying Abstract Interfaces* Report 8815. Washington, D.C.: Naval Research Laboratory.

Coron, P. 1988 — Use of the GALA and PALAS tools to enhance the development of avionic software. *AIAA/IEEE 8th Digital Avionic Systems Conference*. 445-457.

Faulk, Stuart R., and Paul C. Clements 1987 — The NRL Software Cost Reduction (SCR) Requirements Specification Methodology. *Proceedings: Fourth International Workshop on Software Specification and Design*.

Feiler, Peter H. 1986 — *Relationship between IDL and structure editor generation technology* SEI-86-TM-13. Pittsburgh: Software Engineering Institute.

Frakes, W. B. and P. B. Gandel 1989 — *Representing Reusable Software: Survey and Model.* Herndon, Virginia: Software Productivity Consortium.

Funk, Dave 1990 — Letter including "Bofors FS 2000 Application Overview."

Gaffney, John E., Jr. 1990 — *Towards a Mathematical/Information Theory Model of Synthesis.* Herndon, Virginia: Software Productivity Consortium.

Heninger (Britton), K., J. Kallander, D. Parnas, and J. Shore 1978 — *Software Requirements for the A-7E Aircraft.* Naval Research Laboratory, Memorandum Report 3876.

Hester, S. D. Parnas, and D. Utter 1981 — Using documentation as a software design medium. *The Bell System Technical Journal* 60:1941-1977.

Humphrey, Watts S. 1989 — *Managing the Software Process. SEI Series in Software Engineering.* Menlo Park, California: Addison-Wesley.

Integrated Systems, Inc. 1988 — *AutoCode: The "Real-Time" CASE Workbench.*

Jaworski, Allan, Fred Hills, Tom Durek, Stuart Faulk, and John Gaffney 1990 — *A Domain Analysis Process: Interim Report.* Herndon, Virginia: Software Productivity Consortium.

Lamb, D. A. 1988 — *Software Engineering: Planning for Change.* Englewood Cliffs, New Jersey: Prentice Hall.

Lasker, D. M. 1979 — Module structure in an evolving family of real time systems. *Proceedings: Fourth International Conference on Software Engineering.* 22-28.

Lee, Peter 1989 — *Realistic compiler generation.* Cambridge, Massachusetts: MIT Press.

McNicholl, D, C Palmer, and J Mason 1988 — *Common Ada Missile Packages—Phase 2 (CAMP-2)*: Volume III. CAMP Armonics. McDonnell Douglas Astronautics Company.

Meyer, Bertrand 1988 — *Object-oriented Software Construction.* Hertfordshire, U.K.: Prentice-Hall International.

NeXT, Inc — 900 Chesapeake Drive, Redwood City, CA 94063.

Palmer, Constance 1989 — A CAMP Update. In *AIAA Computers in Aerospace VII Conference and Exhibit.*

Parnas, David L. 1972 — On the Criteria to be Used in Decomposing a System into Modules. *Communications of the ACM* 15:1053-1058.

1976 — On the Design and Development of Program Families. *IEEE Transactions on Software Engineering* SE-2:1-9.

1978 Designing software for ease of extension and contraction. *Proceedings: Third International Conference on Software Engineering.*

van Schouwen, A. J. 1990 *The A-7 Requirements Model: Re-examination for Real-Time Systems and an Application Monitoring System* Technical Report 90-276. Queen's University, Kingston, Ontario: Queen's University.

Smith, Douglas R. 1990 *KIDS: A Semi-Automatic Program Development System*. Kestrel Institute.

Software Architecture & Engineering, Inc. 1986 *Introduction to SPECTRUM.*

Software Productivity Consortium 1989a *Dynamics Assessment Toolset User Guide* VAX/VMS Version 1.0. Herndon, Virginia: Software Productivity Consortium.

Software Productivity Consortium 1989b *Dynamics Assessment Toolset User Reference Manual* VAX/VMS Version 1.0. Herndon, Virginia: Software Productivity Consortium.

Szczur, M. 1989 *TAE Plus, A User Interface Development Tool for Building Graphic-oriented Applications*. ACM Symposium, Washington, D.C.

Virtual Prototype Inc. 1987 *Virtual Prototyping Systems*. Montreal, Canada: Virtual Prototypes Inc.

Watts, Richard 1987 *Application generators using fourth-generation languages.* Manchester, England: National Computing Centre Ltd (NCC).

Ward, Paul T. 1989 How to integrate object orientation with structured analysis and design. *IEEE Software*. 74-82.

Winograd, Terry 1979 Beyond Programming Languages. *Communications of the ACM* 22:391-401.

Wolfram, Stephen 1988 *Mathematica: A System for Doing Mathematics by Computer.* Menlo Park, California: Addison-Wesley.

*This page intentionally left blank.*

# BIBLIOGRAPHY


Britton, K.H., R. Alan Parker, and David L. Parnas. *A Procedure for Designing Abstract Interfaces for Device Interface Modules.* Proc. 5ICSE, pp. 195-204, 1981.

Curtis, Bill, Herb Krasner, and Neil Iscoe. "A Field Study of the Software Design Process for Large Systems," *Communications of the ACM* 31 (1988):1268-1287.

D'Ippolito, Richard S. "Using Models in Software Engineering." *TRI-Ada '89 Proceedings*. Pittsburgh, Pennsylvania, (1989):256-264.

Durek, T., C. Brown, D. Hybertson, R. Lutowski, T. Tyler, and F. Van Horne. *Technology Assessment Report: Synthesis*. Herndon, Virginia: Software Productivity Consortium, 1987.

Evans, Michael W. *The Software factory: a fourth generation software engineering environment.* John Wiley & Sons: New York, 1989.

Forman, Ernest H. "Decision Support for Executive Decision Makers." *Information Strategy: The Executive's Journal.* Auerbach Publishers Inc.

Forman, Ernest H. "Executive Decision Support." *Computer Programming Management.* Auerbach Publishers Inc.

Lewis, Ted. "Code Generators." *IEEE Software* (1990):67-70.

Myers, Ware. "Transferring technology is tough." *IEEE Computer* 23 (1990):106-108.

Neighbors, James M. "The Draco Approach to Constructing Software from Reusable Components." *IEEE Transactions on Software Engineering* SE-10 (1984):564-574.

Nejmeh, Brian A. *Characteristics of Integrable Software Tools.* Version 1.0. Herndon, Virginia: Software Productivity Consortium, 1989.

Parnas, David L., Paul C. Clements, and David Weiss. "The Modular Structure of Complex Systems." *IEEE Trans. on Software Eng.* SE-11 (1985):259-266.

Pyster, Arthur. *The Synthesis Process for Software Development.* Herndon, Virginia: Software Productivity Consortium, 1988.

Saaty, Thomas L. *The Analytical Hierarchy Process.* New York: McGraw-Hill, 1980.

Saaty, Thomas L. *Decision Making for Leaders: the Analytical Hierarchy Process for Decisions in a Complex World.* Belmont: Lifetime Learning Publications, 1982.

*This page intentionally left blank.*