An Assessment of Reuse Technology after Ten Years

James M. Neighbors

SADA, 3482 Wimbledon Way, Costa Mesa, CA 92626, USA
neighbrs@netcom.com

Abstract

More than ten years ago the first major workshop on "Reusability in Programming" was held. Since that time some technologies have advanced and come into successful commercial use while others have gone unused. New management and abstraction techniques have aided reuse. Interfacing to huge abstractions, now in common use, has made reuse more difficult. This paper is not a formal survey of reuse technology but instead discusses the evolution of early concepts and the issues they raise. Much of the research of the original workshop participants is just now becoming relevant. In some cases the research of the past points to problems and solutions for the present. As part of this examination the activities in reuse for the next ten years will be forecast and a guide of hard questions to ask purveyors of reuse technology will be provided.

Key words and phrases*: domain analysis, software architecture, software reuse, configuration space, version space, refinement, software component, software process, glue code.*

1. Introduction

The research on software construction from component parts began shortly after it's suggestion by McIlroy [16] at the 1968 NATO conference founding Software Engineering. In the 1970's Software Engineering (SE) research primarily focused on the difficulties of developing systems in the traditional way of new development within large groups. Some early work within Software Engineering [9] attempted to construct systems from large software parts without modification. The resulting systems were brittle and the attempt was abandoned. The Artificial Intelligence (AI) area of automatic programming [2] in the 1970's provided valuable schemes for creating and using abstractions from knowledge bases. The difficulty AI researchers faced was determining what abstraction concepts are useful for the task of programming. SE was struggling to produce that knowledge. By 1980 it was clear that SE and AI were on a collision course in their research paths. The first major workshop on "Reusability in Programming" [21] was held in 1983. As with all strong areas of research the papers show a mix of ideas from different areas. Software Engineering, Artificial Intelligence, and Formal Theory are all represented.

In the intervening ten years there have been surprises. Some technologies such as software libraries and domain analysis have advanced and come into successful commercial use. Other technologies such as transformational implementation and formal theory specification have gone unused. New management techniques (software process and making reuse happen) and new abstraction techniques (packages and objects) have advanced reuse. Huge abstractions have come into common use (graphical user interfaces, databases, networking, multitasking) that complicate reuse which must interface to these abstractions. As expected, it has been shown that reuse enables an organization to build a new version of a system with less effort. This reduces the organization's software development costs. It has also been realized that reuse enables an organization to build a completely new system with less effort. This shortens the organization's time to market and enables the organization to address new markets [13]. This paper primarily focuses on technical tools and techniques rather than management which is co-dependent and equally important.

After ten years of workshops on Software Reusability this is the first conference on the topic. This is an advertisement that Software Reusability has something to offer not only researchers in the area but also practitioners who build commercial software. This paper is not intended as an exhaustive survey of the literature of Software Reusability. There are many good surveys available [6, 11, 12, 25, 26]. Instead I will try to provide a guide to the concepts that have shaped the area, pointing out parts still under construction and parts ready for commercial use. Finally, I will present a set of questions

Keynote address for Third International Conference on Software Reusability, November 1-4 1994, Rio de Janeiro, Brazil. Proceedings to appear IEEE Press.

so that practitioners and researchers can determine for themselves whether of not a technique is mature.

As a structure for this discussion I will use the basic reuse cycle process of Biggerstaff and Richter [5]: Abstraction / Selection / Specialization / Integration. *Abstraction* in this context is how parts are identified, classified, and interrelated. *Selection* is how a part is found for reuse. *Specialization* is how a part may be customized for use in a specific instance. *Integration* is how a part is composed with other parts in a specific instance.

2. Code libraries: a first step

In general I shall use the term *library* to be interchangeable with repository and knowledge-base. *Repository* usually infers that historic and ancillary organization information is kept with the parts to be reused. *Knowledge-base* usually infers that interconnection information is kept between the parts to be reused. I see these as features that all libraries have to different degrees.

The most obvious approach to the problem of software reuse is to form libraries of software modules, but when we consider the reuse of existing programs we must be careful in describing the goals of the reuse. In *black-box reuse* a programmer is looking for a program part that can be "plugged in" without modification. In *white-box reuse* the programmer is looking for a program part that can be modified before use. This is an important consideration in the design of a library of reusable program parts. Black-box reuse libraries need only be concerned with what the program part does. White-box reuse libraries must store both what the part does and how it does it.

Black-box reuse has been successful for many years as run-time libraries and mathematical libraries. In both cases the data structures manipulated by the library parts have either been controlled as with the compiler run-time libraries or very basic as with the math libraries. White-box reuse has been successful as collections of algorithms with detailed descriptions of their operation above the level of programming language code.

Black-box library systems must overcome two basic problems of classification and search. The *classification problem* of how to describe what each part in the library does. The *search problem* of how to find parts in the library that address your problem. White-box library systems must overcome two additional problems of structural specification and flexibility. The *structural specification problem* of how to describe how each part in the library works. The *flexibility specification problem* of how to define the decisions inherent in the parts and the constraints upon the composition of parts.

Related to the issue of white-box or black-box reuse is the issue of *granularity of reuse*. *Large-grain reuse* implies that the parts stored in the library are usually very large, perhaps the equivalent of thousands or millions of lines of code. On the other hand *small-grain reuse* implies that the parts are usually very small, perhaps only a few to a hundred lines of code.

The overall *library problem* is the conflict between the goals of reuse and the structure of the library. If the parts in the library are to be simply reused without modification (black-box reuse) then they must be large and somewhat inflexible to support standard interfaces (large-grain reuse). However, if the individual parts in the library are large (complex structural and flexibility) then the number of parts may be small (simple classification and search).

If the parts in the library are to be modified and reused (white-box reuse) then they must be small to be general, flexible, and understandable (small-grain reuse). However, if the individual parts in the library are small (simple structural) then the number of parts in a usable library must be very large (complex classification, search and flexibility).

Thus the objectives of black-box reuse and white-box reuse are always in conflict. If a library contains many small parts, then it lessens the structural specification and intra-part flexibility problems at the expense of increasing the inter-part flexibility, classification and searching problems. If a library contains a small number of large parts then it lessens the classification and searching problems at the expense of increasing the structural specification and flexibility problems.

In the last ten years a great deal of work has been done with libraries for black-box reuse [23]. They have been successfully deployed on commercial projects and improvements of 15% and higher have been reported [13]. The lesson here for a practitioner is that if your organization has not already constructed a library then do so now. It is a good first step. The technical and managerial experience is available to make it a success. As will be discussed later, we have also learned that there are limits to libraries supporting only the black-box reuse of program code.

A truly great library must support both white-box and black-box reuse. At the minimum we must have black-box reuse for interfaces to parts of the developing system that the developers have no control over, such as the huge abstractions of the 1980s GUIs, DBs, OSs, and networks.

3. What to reuse

Software Engineering has studied the process of building software for the last twenty-five years. Briefly the following lifecycle phases and work products are important. Each phase is annotated with an estimate [8] of the percentage of the total effort to produce a first system.

- *Requirements* (6%) provide the function, performance, and external constraints on the system.
- *Analysis* (8%) interrelates the data, function and external interfaces to ensure that the requirements are complete and understood.
- *Design* (29%) casts the understanding of analysis into an architecture of computer process structures and details the function of members of the structures.
- *Implementation* (34%) constructs actual program code and tests individual codes.
- *Testing* (23%) checks compositions of codes and functional performance of the resulting system.

Maintenance is a reiteration of these same activities. All the phases are accompanied by *simulation* and management. Sometimes lifecycle phases are confused with the basic styles of *management* given below.

- *Waterfall* management performs each phase completely before beginning the next phase.
- *Inspection* management examines the results of each phase and falls back one or more phases if there is a problem.
- *Spiral* management performs each phase lightly and then reiterates for a richer system behavior.

The implementation phase consumes the largest percentage of the effort in building a system. As such it is a prime candidate for automation. However, consider the consequences. Using a technique such as reuse to completely eliminate implementation would mean only a 34% savings in effort. This estimate discounts any extra effort involved in using the technique. Still, as I have mentioned before, a potential 34% savings should not be passed up. The construction of a black-box reuse code library is well worth the effort.

A library of programming language level parts for reuse can have a huge impact on implementation but very little impact on the other phases of the lifecycle. However, since implementations are derived from designs, designs from analysis, and analysis from requirements, the earliest phase information we can store in our library will have the greatest impact on later phases. This will be especially true if we provide interconnections to partial work products of later phases. The problem with this approach is that the work products of the higher phases tend to be very specific to the problem domain. All of the library problems discussed in the earlier section on code libraries are independent of the work product stored in the library. Thus a library that stores analysis, design, and code information inherits all the problems of a simple code library.

4. Using the library

Typically one thinks of the components of a library as perhaps related but not explicitly connected. Like a book library, each component is expected to stand on it's own. The abstraction took place when the author placed the content into a container (book) and then specified the content to a separate content description scheme (catalog). A problem with this scheme is that it places the full burden of using the library on each individual user of the library. In the basic cycle of reuse this means that the author performs the abstraction step while each and every user performs the selection, specialization and integration steps. Obviously if we could provide a scheme where some of this burden were shifted back to the author / abstractor then it would have a significant impact. I will discuss approaches to this in the section on domain-specific knowledge.

Once a user has selected two reusable parts - either code or a higher phase work product - then it is left to each user to make these two parts work together. Integration in this context is *composition* of the two components. One approach to this is to write or synthesize glue code. *Glue code* transforms the abstractions of each component and results in compatibility. It does this by forming a model of compatibility, implementing it, and embedding the components to be interfaced. Once again each user of the library must do this.

As a large system is constructed from reusable parts embedded in layer upon layer of glue code, it becomes hard to see the components for the glue [7]. In some cases the abstractions may just not be compatible by any amount of glue [4]. Ultimately glue is most of the system code. Since glue is custom to each user of the library, not much has be gained by reuse.

Current very large hand-written systems contain huge amounts of glue. Glue accumulates through the process of maintenance. Consider what "glue" actually is - an abstraction created to bind together two separate abstractions. Maintenance programmers on a large system will avoid modifying glue sections because they bridge two abstractions. The possibility of introducing an error with a maintenance change to glue is high because the content coupling is very high. Maintenance programmers will tend isolate their changes by surrounding them with complex predicates to limit the impact of the change to the system function. These predicates are just more glue. Consider what a system looks like after 10 years of this

kind of maintenance. Ultimately huge sections of the code are dead, even though they are strongly connected, because the conjunction of maintenance predicates to reach much of the code is never true. There is always some glue, but it should not be the basis of construction.

5. Domain-specific knowledge

One approach to easing the selection and integration problems is to be problem domain specific. In this approach the library uses the existence of problem domains as an organizational scheme. The work of Batory et al. on database system generation is a nice example [3]. The knowledge about database generation could have been presented in three basic ways.

1. A library full of components for all kinds of domains - e.g., graphics, networking, database. You find the components you need to put together a database and write glue to compose them.
2. A library full of database components. You find them. You figure out how they fit together. You compose them.
3. A library full of components, some of which are database components. The library also contains interconnections between the components that provides an architecture or template for how to fit the components together to form a database. You fill in the architecture with your selections.

Clearly for users the third form of library is preferable. The selection and integration effort are guided by information added by the author at abstraction time. Specialization guidance can also be added. This requires the abstract creator to analyze the problem domain, which is called *domain analysis.* The abstract creators job becomes much harder. Domain analysis differs from classical system analysis in that an entire family of systems and solutions must be considered. It has been intensively studied in the last ten years [1, 24]. The process of domain analysis is useful for an organization even if it is not used in a reuse scheme. It characterizes the kinds of systems the organization builds and that in itself is a valuable educational tool.

Briefly an approach to domain analysis is as follows:

1. Analyze four or more existing systems in the domain using classical systems analysis [17].
2. Form an analysis model of the union of features of existing systems.
3. Determine and, or, not, existential and universal constraints on acceptable domain feature variations.
4. Present model and variation constraints to domain experts for approval.
5. Continue 1 through 4 until all systems in 1 are variants of model 2 under constraints 3 and approval 4.
6. Provide function implementations under allowed variations.

The topic is much more complex than this; examine the Prieto-Diaz and Arango tutorial [24] for a detailed discussion.

In the brief discussion above I was not very precise as to the specific result of domain analysis. The work product of domain analysis is not well defined because there are many approaches to using domain-specific knowledge. Some system specification techniques using domain knowledge are given below.

- *Domain languages* [19] constrain system descriptions to legal statements in an ad hoc source form language.
- *Domain algebra* [28] constrain system descriptions to legal statements in a high-level formal algebra.
- *Domain-specific software architectures* [15] constrain system descriptions to a single architecture template to be elaborated with detail.
- *Domain-specific kits* [13] provide components, frameworks, glue languages, generic applications, and tools.

The existence of the work products above and those from early lifecycle phases are stretching our use of the term library for where we keep all this information [13].

6. Abstraction and refinement

The act of analyzing and abstracting a problem domain explicitly connects library components. As discussed earlier, these components may be code, immutable black-box interface specifications, and high-level domain-specific specifications. Our simple code library is beginning to look like a complex knowledge base. As with a simple code library where the user needs help finding a component, the user of a knowledge base needs help to navigate through all the possible system solutions.

For each component in a given library I make the following definitions.

- *level of abstraction (LOA)* - on a per component basis is the minimum number of decisions to reach any compilable state by a commercial tool.
- *level of refinement (LOR)* - on a per component basis is the number of decisions that have been made to arrive at this component.

Domain-specific interconnections may cause decisions on one component to constrain components to which it is connected. Thus one decision may increase or decrease the level of abstraction for a collection of

connected components. *Refinement* is the process of making these implementation decisions. There are two basic approaches to refinement. A component can refine directly to code (LOA = 0) or it can refine to one or more connected components that require further refinement (LOA > 0).

As refinement of a complete system proceeds from initial system specification to code the following effects are found [18].

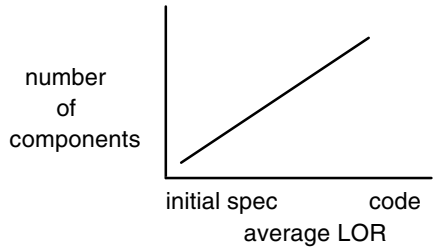

Figure 1. Components vs. average LOI

In general, the refinement of higher LOA components produces more lower LOA components (Fig. 1). These lower LOA components must be further refined until they are all code components (Fig 2.).

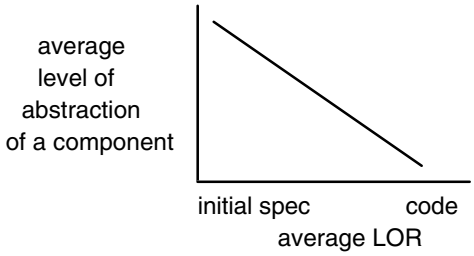

Figure 2. average LOA vs. average LOI

Using the definition of LOA as decisions to be made and the general shapes of Figures 1 and 2, the number of decisions pending can be estimated (Fig. 3).

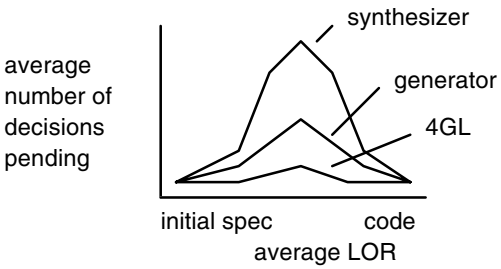

Figure 3. Decisions vs. average LOR

The peak in the center of Figure 3 is the *intermediate modeling swell*. Initially the number of decisions grows rapidly as refinement implementation decisions are made. The number of decisions then falls off rapidly as previously made decisions severely constrain later choices. The actual shape of Figure 3 is not as important as the idea that in some cases there is a large space of decisions and constraints to navigate.

The family of curves shown in Figure 3 represents the complexity of the refinement. Each curve is annotated with a class of *refinement mechanism* capable of navigating the decision space. *4GLs* have the lowest curves, *program generators* the next highest, and *program synthesizers* the highest.

It is important to recognize that power to handle complex refinements does not infer the power to construct powerful systems. The converse is also true. Simple refinement mechanisms are capable of building powerful systems. A powerful refinement mechanism simply provides for richer system variations and higher LOA components. Macro expansion can serve as the refinement mechanism for program generators. AI problem solving and theorem proving is used as the refinement mechanism for program synthesizers. The AI problem solving is necessary to navigate through the large space of pending decisions and their interactions.

7. Structural architecture from refinement

In the above exposition the ultimate destination of refinement is "code". This is a bit simple since large systems are not simply made of "code".

During refinement there are two major forces at work. One force, stepwise refinement [30], wants to further divide each component into collections of mostly lower level components. Division serves to better define the function of the original component and provides the *functional architecture* of the developing system by elaborating what function the system performs. Stepwise refinement is analogous to the basic AI problem solving method of divide and conquer.

The other force, structural encapsulation, wants to encapsulate the concept embodied by the component in a structure that will be visible in the resulting code. The creation of a named callable routine is an example. The body of a routine can always be instantiated on its arguments and inserted inline for the call. For reasons having little to do with the specific problem domain at hand, structural encapsulations are performed for clarity and/or efficiency. These encapsulations establish the *structural architecture* of the system. In Software Engineering this force is roughly analogous to creating systems from layers of virtual machines [10]. In AI this is similar to the set-of-support strategy. The decision as to which force prevails in a certain case is maximal "information hiding" [20]. This concept is further

resolved to minimizing coupling and maximizing cohesion.

The dynamic between functional architecture and structural architecture is nothing new. As with building architecture, Louis Henry Sullivan's declaration "outward form ever follows function" [29] advocates functional decomposition. R. Buckminster Fuller's Dymaxion principle [14] states that architectural designs should stress deriving maximum output from minimum material and energy. This can be taken as a scheme of maximum structural concept encapsulation. His geodesic dome design of only hexagons and pentagons epitomizes this approach. Note that these forces are not always in opposition. In many cases they are in support of each other. The same is true of the functional and structural architectures of information systems. Object-oriented technology has had a high impact because it strongly supports both functional and structural architecture.

As a system is maintained and upgraded changes occur to both the functional and structural architecture. Changes in the functional architecture create new entries in the *version space* of a system. These typically include adding more functional features to the system. Changes in the structural architecture create new entries in the *configuration space* of the system. These typically include adding support for new compiler, hardware and operating environments. All concrete actual systems are an entry in the version by configuration space. As an example, the system might be Microsoft Word; the version might be v2.0; and the configuration might be for Microsoft Windows v3.11 using Microsoft C v6.0. *Configuration management* attempts to manage this space.

Let us consider the structural architecture refinement possibilities for a very simple functional architecture: a sine routine that takes a number in radians and returns a number between 0 and 1. This simple functional architecture serves to focus our attention on the structural architecture. Some legitimate structural refinements include:

1. Encapsulate sine as a system global function with passed read-only argument parameter and return value on the stack. Provide functional call mechanism and naming as resources. Use existing sine function if compatible otherwise recursively represent sine as function to be constructed. Annotate module interconnection language (MIL) [22] to indicate required access to the function. Add created function to construction dependency list for the current component.
2. Encapsulate radian as an object and derive sine as a method. Provide object mechanism and object naming. Use existing object type and method if available, otherwise recursively represent sine as method to be constructed. Modify MIL and dependency list as above.
3. Expand sine function semantics inline. Provide sine semantics, local variable allocation and naming. No MIL or dependency list modification necessary.
4. Encapsulate sine as a table interpolation function given table and radian value, return sine value on the stack
5. Encapsulate table interpolation function to use in-memory tables computed at system start and destroyed at system finish
6. Expand table interpolation function inline to use in-memory tables loaded from a file at system start and destroyed at system finish. Provide connection to system start and finish chains, memory allocation, file operations, location of files at runtime. Annotate MIL to require construction of file at system construction time. Add file to system distribution.

The goal here is not to create a complete list but to establish that the list is rich. All of the above schemes use actual programming language code. Some schemes, like threaded code and threaded code interpreters, can blur this line even more.

Real systems exhibit these behaviors. Consider configuration files which most programs use and require defaults. Configuration files are an example of the last structural architecture listed above. Consider the processes assumed and work products created: MILs, dependency lists, system distribution lists, programs executed at system creation time, compilations at system creation time, system execution start, system execution, system execution finish. These are what real systems are made of. What manages these with reusable software? Very little work has been done in this area of developing structural architectures during refinement.

8. Reuse technology questions

Some basic questions should be asked about a reuse technology before its adoption. These questions have no "right" answers but are more a checklist to confirm that the concepts have been considered by the users and suppliers of the technology.

Organizational

1. How does this scheme fit into my organization?
2. How is my version space impacted by using this method?
3. How is my configuration space impacted by using this method?
4. How is the basic reuse cycle of abstraction, selection, specialization, and integration performed?

Specification of a system

1. What is the form of specification?
2. What is the LOA of the starting specification?
3. What is the result of domain analysis?

Refining specification to implementation

1. How is a consistent implementation maintained as the LOR increases?
2. How does the method / tool support various functional architectures?
3. How does the method / tool support various structural architectures?
4. How is specialization performed?
5. How do maintenance programmers maintain?
6. How do I guide refinement to different targets (e.g., implementations and simulations)?

Extending the library/knowledge base

1. How is the library/knowledge base extended?
2. How are new subsystem interfaces (e.g., GUIs, DBs, Networks, OSs) specified?

9. Conclusions and the next 10 years

In the last ten years we have successfully constructed and used general code libraries. They have proven to be quite successful. We have also investigated and used the power of domain-specific knowledge in various approaches. It has proven to be even more powerful. These techniques were only of use because the organizational dynamics were studied and the ability to make organizational change developed.

Currently the emphasis in reuse has shifted away from code libraries toward knowledge bases containing problem domain specific information. The mechanisms for using these knowledge bases to create systems with rich functional and structural architectures hasn't been explored recently. The original "Reusability in Programming" Proceedings [21] had quite a few papers on *transformational implementation,* but not much progress has been made in this area in the last 10 years. I would expect a fair amount of research to be done in this area in the next 10 years. At the same time powerful commercial systems will be made with refinement mechanisms much simpler than transformational implementation.

It is tempting to separate systems by their interfaces and hypothesize cooperating communities of processes that co-exist and provide function for one another. We "plumb" these large autonomous pieces together using passes, pipes and network protocols. This use looks like a perfect example of black-box reuse. What happens in the long run? The same thing that happens when we glue two program fragments together. In this case the problem is made worse because the three separate program parts (the two cooperating software agents and information carrying agent) all have their own version and configuration spaces. As each of the separate parts is maintained, a little bit more glue must be added to maintain compatibility with the other parts. Thus the glue code itself has a huge version space. Ultimately we have such large amounts of glue and such complex version/configuration spaces that performance degrades and the overall system becomes bug ridden. We have done this before. This kind of interconnection is analogous to the proliferation of JCL interconnections on mainframes in the 1970s. Vendors sold large-grain software components such as ISAM and sorts. The software systems became structurally brittle, functionally inflexible, and very expensive to run. This would be an expensive mistake to make again. In the future I would expect reuse method developers to carefully explain how their method deals with maintenance and the version/configuration space over time.

All the systems that we build have to be composed with other complex systems. At a minimum they are composed with the computing hardware and its compiler. These complex systems have immutable functional and structural architectures. Bindings to them must be able to describe these rich architectures. Refinement mechanisms must be able to both use these binding descriptions and create various functional and structural architectures for a given system specification. A system refinement should result in much more than just code.

Program construction is much easier than program understanding, since a program constructor is given a knowledge base from which to create programs by navigating a space of possibilities. When the domain-specific knowledge of many domains is added to that knowledge base, the range of possibilities becomes practically unbounded. A program constructor only has to navigate the given space. A program understanding system has to recreate the entire space and recognize program features in it. Similarly component customization is much easier than composing two components. A component that refines into two lower level components provides each user with a space to navigate. Any glue that must be specified is provided by the abstractor when the component and its refinement connections are added to the knowledge base. We cannot really expect programmers to create glue for domain-specific components each time they are used. Thus a scheme of domain-specific specification with knowledge-based refinement where maintenance is performed by re-refining is necessary. Much work needs to be done in this area.

All of the above discussion can be directly applied to the implementation of organizational systems as opposed to software systems. As an example, the accumulation of

maintenance glue in software systems is analogous to the expansion of bureaucracy in organizational systems. The currently popular "Re-engineering the Organization" is classical systems analysis applied to existing organizations to remove glue. Some organizational domains have been analyzed, such as the software system factory [27]. Organizational processes and technical development processes are co-dependent. It is only natural that in the future we would require the reuse of one kind of process to reuse the associated other kind of process.

Finally, if we adopt a stance of top-down synthesis from problem domain-specific knowledge rather than bottom-up composition of code, then we have achieved a true breakthrough - we no longer have to generate just implementation code. We can generate code for different reasons. We can also generate hardware architectures using hardware description languages such as VHDL. As an example, consider a single network protocol description that can be refined into the following systems:

- Software implementation of the protocol.
- Software simulation of the protocol.
- Software/hardware implementation of the protocol.
- Hardware test bench for the protocol.

Guiding the refinement to different goals is an open research issue. We have come a long way and we have a long way to go.

References

[1] Arango, G., and Blum, B., Special Issue on Applications of Domain Modeling to Software Construction, **Intl. Journal of Sfw. Eng. and Knowledge Engineering**, Vol. 2, No. 3, September, World Sci. Pub., 1992.

[2] Balzer, R., A 15 Year Perspective on Automatic Programming, **IEEE Trans. Sfw. Eng**., SE-11, pp.1257-1268, November, 1985.

[3] Batory, D. et. al., GENESIS: An Extensible Database Management System, **IEEE Trans of Sfw. Eng.**, SE-14, pp. 1711-1730, November, 1988.

[4] Berlin, L., When Objects Collide, **Proceedings OOPSLA90**, pp. 181-193, ACM Press, 1990.

[5] Biggerstaff, T., and Richter, C., Reusability Framework, Assessment, and Directions, **IEEE Software**, Vol. 4, No. 2, pp. 41-49, March, 1987.

[6] Biggerstaff, T., and Perlis, A., eds., **Software Reusability**, volumes 1 and 2, ACM Press Frontier Series, Addison-Wesley, 1989.

[7] Biggerstaff, T., An Assessment and Analysis of Software Reuse, **Advances in Computers**, Vol. 34., Academic Press, 1992.

[8] Boehm, B.W., **Software Engineering Economics**, pp. 66, Prentice-Hall, 1981.

[9] Corwin, W., and Wulf, W., SL-230: A Software Laboratory Intermmediate Report, Technical Report, Carnegie-Mellon University, May 1972.

[10] Dijkstra, E., Complexity Controlled by Hierarchical Ordering of Function and Variability, in **Software Engineering**, Naur, P. and Randell, B., eds., NATO Science Committee Report, pp. 181-185, Germany, 1968.

[11] Freeman, P., ed., **Tutorial: Software Reusability**, IEEE Press, 1987.

[12] Krueger, C.W., Software Reuse, **ACM Computing Surveys**, Vol. 24, No. 2, pp. 131-183, June, 1992.

[13] Griss, M.L., Software Reuse: From Library to Factory, **IBM Systems Journal**, Vol. 32, No. 4, pp. 548-566, 1993.

[14] Marks, R.W., **The Dymaxion World of Buckminster Fuller**, So. Illinois Univ. Press, 1960.

[15] Mettala, E., The Domain-Specific Software Architecture Program, Special Report CMU/SEI-92-SR-9, CMU Software Engineering Institute, June 1992.

[16] McIlroy, D., Mass Produced Software Components, in **Software Engineering**, Naur, P., and Randell, B., eds., NATO Science Committee Report, pp. 138-155, Germany, 1968.

[17] Natl. Inst. of Standards and Technology (USA), **Integration Definition for Functional Modeling (IDEF0)**, FIPS standard 183, December, 1993.

[18] Neighbors, J.M., **Software Construction using Components**, Ph.D. dissertation, University of California, Irvine, May, 1980.

[19] Neighbors, J.M., Draco: A Method for Engineering Reusable Software Systems, pp 295-319 in [6].

[20] Parnas, D., On the Criteria to be Used in Decomposing Systems into Modules, **Comm. ACM**, Vol. 15, No. 12, pp. 1053-1058, 1971.

[21] Perlis, A., **Proceedings of Workshop on Reusability in Programming**, Newport, RI, September, 1983.

[22] Prieto-Diaz, R., and Neighbors, J.M., Module Interconnection Languages, **Journal of Systems and Software**, Vol. 6, pp. 307-334, 1986.

[23] Prieto-Diaz, R., and Freeman, P., Classifying Software for Reusability, **IEEE Software**, pp. 6-16, January 1987.

[24] Prieto-Diaz, R. and Arango, G.,eds., **Domain Analysis and Software Systems Modeling**, IEEE Computer Society Press, 1991.

[25] Prieto-Diaz, R., Status Report: Software Reusability, **IEEE Software**, pp. 61-66, May, 1993.

[26] Tracz, W., ed., **Tutorial: Software Reuse: Emerging Technology**, IEEE Press, 1988.

[27] Scacchi, W., The Software Infrastructure for a Distributed System Factory, **Software Engineering Journal**, Vol. 6, No. 5, pp. 355-369, September, 1991.

[28] Srinivas, Y., **Algebraic Specification: Syntax, Semantics, Structure**, Technical Report UCI-ICS-90-15, ICS Dept., University of Calif., Irvine, 1990.

[29] Sullivan, H., The Tall Office Building Artistically Considered, **Lippincott's Magazine**, March 1896.

[30] Wirth, N., Program Development by Stepwise Refinement, **Comm. ACM**, Vol. 14, No. 4, pp. 221-227, 1971.