LA-UR- 00-758

Approved for public release; distribution is unlimited.

Title: Software Technology for Composition-Based Simulation Construction A Roadmap

Author(s): Randy E. Michelsen
J. Wayne Anderson
Joe V. Holland

Submitted to: Defense Advance Research Projects Agencies

Los Alamos
NATIONAL LABORATORY

Software Technology

for

Composition-Based Simulation Construction

A Roadmap

The reuse of software has been an eagerly sought yet elusive goal of software developers for decades. Many promising technical approaches have been pursued, but none have yielded the full potential. Recent advances in software technology R&D have led to substantial progress in the pursuit of software reusability. One of the more promising avenues of work is composition-based (component-based) software construction. This paper explores the compositional approach to the construction of software in general and presents a "technology roadmap" for software composition. This roadmap depicts the interdependencies and state of the technologies necessary to fully realize software composition.

Introduction

The rapid and pervasive growth of computing technologies in the last decade, highlighted by the emergence of the Internet, has fundamentally altered the nature and speed of the evolution of software technologies. For example, business process reengineering necessitated by the rapidly changing business environment has increased the need for flexibility and extensibility in related software products. In partial response, the topic of software reuse is being aggressively explored in both academic and industrial settings. One promising reuse approach is centered on techniques for the composition of a software system from well-defined, individual software *components*.

The traditional process for developing large-scale software, including computer-based simulations, is cumbersome, time consuming, costly, and, measured in terms of the flexibility of the product, generally inadequate. Composition-based development approaches are designed to reduce the development cost and time for individual software products, while still yielding a product that is more maintainable and extensible. In such systems, the elementary unit of composition (component) is generally a well delineated, relatively independent, and replaceable part of a software system performing a specific function. While composition-based software construction offers obvious benefits, intuitively the identification of appropriate compositions simply derived from the complete set of all possible components is intractable [1] in the general case. Though not surprising, this supports the focus of current research activity in composable software systems on approaches to describe or constrain the set of "valid" compositions.

Many researchers are studying component-based approaches to software development in general, and a few have focused specifically on simulations. In a component-based approach for the development of a simulation, functional or logical elements of simulation entities are represented as coherent collections of one or more components satisfying explicitly defined interface requirements. A simulation is a top-level aggregate comprised of a collection of components, each component representing an individual simulation entity, that interact with each other within the context of a simulated environment. A component may represent a simulation artifact (e.g., run-time data collectors), an agent, or any entity that can generate events affecting itself, other simulated entities, or the state of the system. The component-based approach promotes code reuse, contributes to reducing time spent validating or verifying models, and promises to reduce the cost of development while still delivering tailored simulations specific to analysis questions.

This paper defines a technology roadmap for software composition considering technologies in the broader software development context. The rationale is that most if not all of these technical issues are directly relevant to the more narrowly focused simulation community. The roadmap therefore presents the technologies required to realize the capability in a general setting, though the underlying motivation for the study is understanding composition in the context of large-scale computer simulation construction. The roadmap does not include implementation plans or performance targets. These elements, though often included in typical technology roadmaps, were considered beyond the scope of the current effort. The assessments of technology areas represented in the roadmap were based on applicability to broad use in the software development community. The roadmap remains relevant since the required technologies are not dependent on the domain of application. In the case of simulation construction, a "successful" system would enable an individual, whether an experienced software developer or simply a knowledgeable end user, to assemble a simulation application from existing software components in a relatively short time. The paper concludes with an appendix briefly describing the 1995 report of the DoD Software Reuse Initiative and a bibliography of relevant literature from the general software community published primarily since that time.

The End-Use Context

Though composition as a development approach is broadly applicable, the focus of the study was the supporting technology for composition of large-scale computer simulations. A number of end-use scenarios [2] defining operational characteristics of the future capability in specific contexts where developed to provide functional requirements for a composition capability. The scenarios were defined in operational terms and included the definition of specific capability-oriented issues. Subsequently, these end-use scenarios guided the specification of an abstract use case describing the general characteristics of the projected end-use context.

End-Use Scenario A: Simulation-Based Training Exercise

The current process for the scenario generation and planning of an exercise for a JTF Commander and his staff can take over a year. The operational requirement exemplified by this scenario is a Commander completing the planning process for an exercise in less than four hours. At the end of this four-hour period, the system would compose the simulation system and supporting initialization files (synthetic natural environment data, initial conditions for the simulated entities, force positioning, etc.). Representational and technological issues to be addressed in this use case include:

- Missing or new equipment that would require new doctrine and tactics;
- Ability to specify interfaces to C4I interfaces;
- The exercise may include coalition forces and equipment;
- The exercise may include civilian forces (Red Cross, Department of State, Department of Transportation, etc.);
- Changes to the command and control structure that may need to occur; and
- The use of this technique in a closed form solution versus man-in-the-loop for training.

End-Use Scenario B: Advanced Systems Concept Exploration

Approaches for simulation-based advanced concept development generally postulate a computational environment supporting the exploration of various technological elements in a new system design. An example might be exploring the utility of a tank incorporating a hovercraft-like capability. For such an approach to be useful the construction of the computational representation should ideally require only a few hours and the resulting simulation should support the ready exploration of alternatives. The simulation environment must enable the study of issues such as:

- The development and fielded cost of the addition of this new system or capability;
- Benefits of the capability represented by the concept; and
- Impact on doctrine and tactics.

End-Use Scenario C: Course of Action Analysis

Decision-makers are in need of course of action analysis tools to assist them in making well-informed and timely decisions in complex situations. An example of such a situation would be the occupation of a US Embassy by an unknown terrorist organization. The organization has threatened to detonate weapons strategically placed in major cities around the world if the Embassy is stormed. In this use case, the on-site military commander needs to understand the risks associated with the various courses of action available. Because of the time-critical nature, the analysis must be completed in a short time and provide a credible representation of the various courses of action to be assessed. Issues of interest in this context include:

- Visualization of information of a non-spatial nature; and
- Understanding risk associated with specific courses of action.

End-Use Scenario D: Force Structure / Command & Control Concept Exploration

Traditional force structure analyses often require an assessment of the impact of organization change with respect to mission effectiveness or other measures. For example, assume the US Army is studying the removal of Brigade from the Army echelons. To understand the benefits and disadvantages of this action, the analysts need an environment that permits them to simulation Army operations without Brigades. Issues to be explored in this use case include:

- Impact of decision making at lower echelons;
- Impact on planning at higher echelons;
- New information requirements; and
- Command and Control requirements (new equipment, better communication, etc.).

End-Use Scenario E: Simulation Composition using Shared Components

In this end-use scenario, two concurrent study teams are developing simulations using one or more shared elements (e.g., an advanced armored vehicle concept study and a future force projection requirements study). The temporal aspects of the sharing are such that the simulation representations employed in one of the studies are to be incorporated into the second study. Issues to be explored include:

- Semantics of the sharing (e.g., degree of coupling)
- Control/coordination of sharing; and

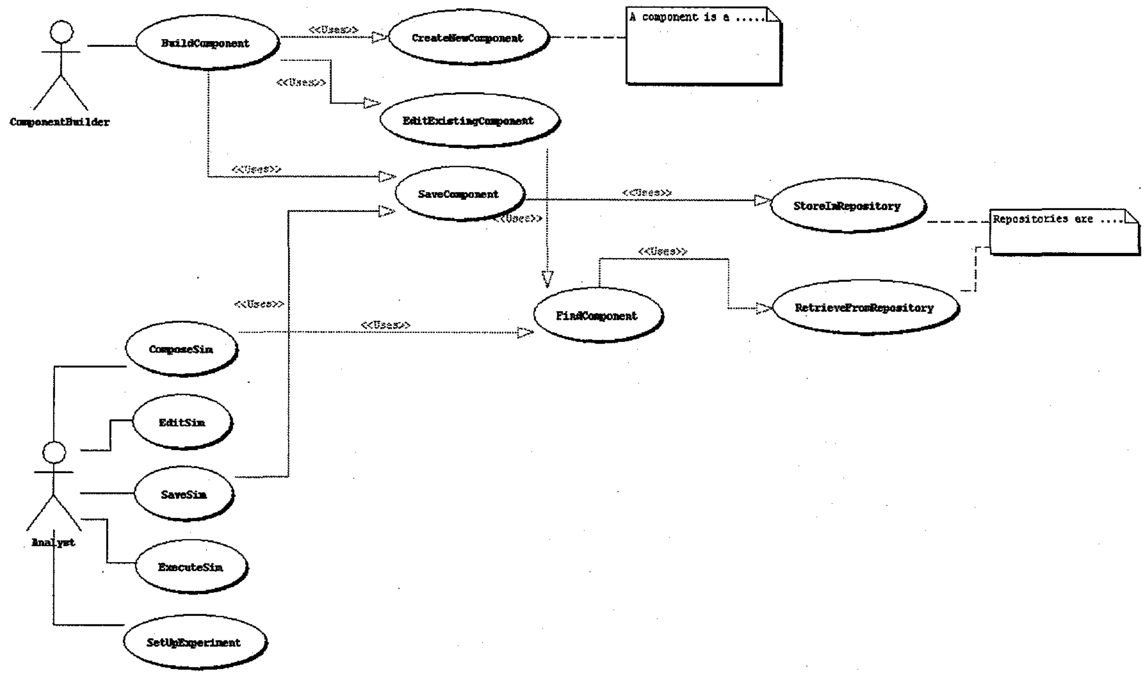
- Framework infrastructure implications

These five end-use scenarios are representative of the broad range of applications in which simulation composition can be employed. Each of the scenarios has a temporal element that, while manifested in a different operational setting, exceeds the capabilities of current computer simulation environments.

Abstract Use Case

The underlying goal of a simulation composition capability is to "Build the simulation to fit the analysis problem". Ideally, this must incorporate a very extensive set of reusable and modifiable components that are shared across a large number of organizational and geographical boundaries. The abstract use case is intended to represent this end-use.

Figure 1 Abstract Use Case Diagram



Two end-use venues, static and dynamic composition, can be generally defined for composition-based construction. Though represented by the abstract use case, they present different requirements for the composition process and have far ranging implications for the underlying technological support. A static composition occurs in a development context in which component collections ("libraries" in the traditional vernacular) are used in the construction of an application instance comprised of a group of components "bound together" to create a static application. In this setting, the identities of the individual components are subsumed by the identity of the newly formed application (component). In contrast, dynamic composition occurs in a development context in which collections of shared components are used to create an application comprised of linked components. Linked components retain their identities while the newly created application (component) has an identity in its own right.

The Technology Roadmap

The technology roadmap depicts the technological areas required to support the construction of software systems through composition. The application focus is the domain of computer-based simulations, though the required technologies are not generally application domain specific.

The essential structure of the roadmap depicts functionally related groupings of technologies, represented as nodes in the roadmap, and their relationships, modeled as arcs, to one another. The roadmap has an inherently hierarchical nature related to the interdependencies among the technologies. It may also be viewed as an overlay associating technologies with the capabilities referenced in the abstract use case.

The basic elements of the roadmap are functionally related groupings of technologies. The most fundamental such elements are:

- *Component Creation* – the initial definition and development of components;
- *Component Management* – the collection and maintenance of a set of components such that they may be effectively reused; and
- *Component Utilization* – the technology elements that support the effective reuse of individual components.

Each of these elements is represented by a number of related technologies. For example, *Component Utilization* is comprised of technological elements including *Component Interface Models* and *Data Exchange Models*. The former include mechanisms by which components reveal their command and functional interfaces, e.g., the Java reflection application programming interface, adaptable component interfaces, and component interconnection models. The latter defines the low-level data exchange/transport mechanism between two "connected components".

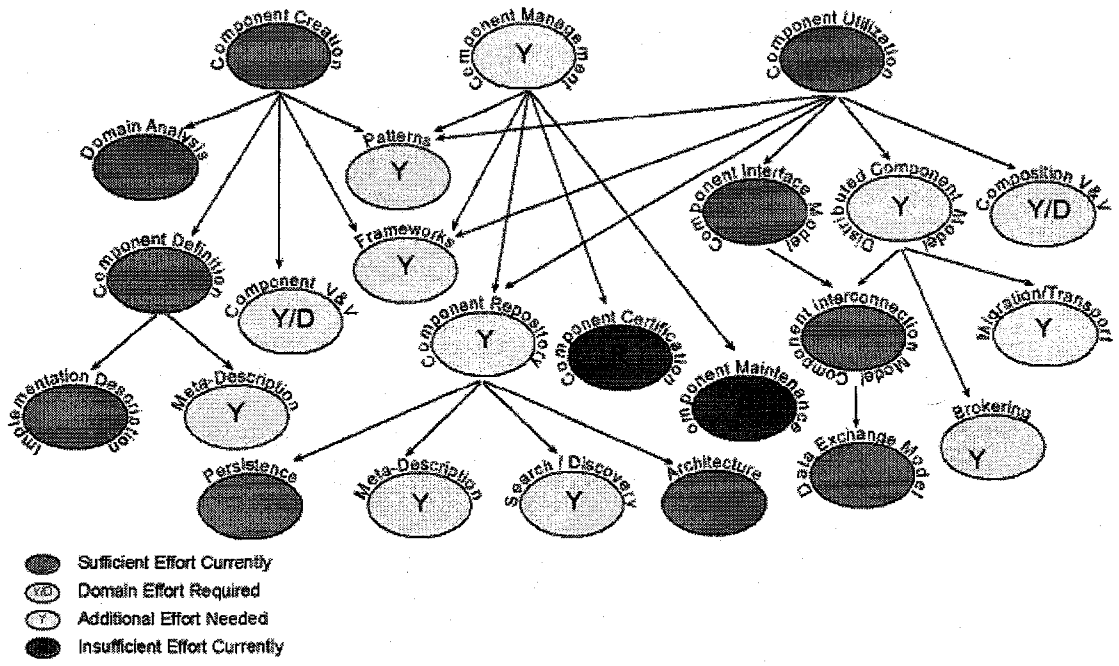


Figure 2

The Roadmap.

The roadmap uses color-coding, augmented with a character designation, to indicate the current state and vitality of the associated technology areas with respect to the realization of software composition. Green is used to designate technology areas in which there is presently sufficient R&D results or activity in the software community. Nodes are colored yellow ("Y") to indicate areas in which there is substantial activity but additional effort is required. Several such technology areas are further designated "Y/D" to indicate that effort focused specifically on simulation issues are required. Several technology areas are colored red ("R") indicating a lack of evidence of sustained, substantive R&D efforts. The following sections discuss individual technology areas that require additional effort.

Meta-Description

Meta-description refers to the representation of a component (or system) by a formal mechanism describing its semantic properties. For example, specification languages based on formalisms such as first order predicate calculus, temporal logic and state machines can be used to formally specify hardware and software systems [3, 4]. These specifications may in themselves be executable, allowing system developers to treat the specifications as prototypes [3]. These formally based descriptions lend themselves to analysis of component and system characteristics.

Formal mechanisms have been developed for general software systems (e.g., Z [4], Communicating Sequential Processes) as well as, more specifically, for larger scale software architectures (e.g., UniCon [5], Rapide [6]). Though still the focus of substantial research efforts, specification languages have been successfully used to formally specify relatively small systems or parts of systems.

There are a number of promising avenues for future work in this area. Successful component specification formalisms would enable semantic-based component classification and repository search, as well as compositional strategies incorporating reasoning about component behaviors. These would lead to substantial improvement in component reusability. Further research may lead to the general applicability of this technology in the formal definition of software components and composed systems.

Verification and Validation

Verification and validation (V&V) are processes used to ensure that software being developed meets design requirements and performs correctly. Software verification refers to the process of ensuring that each step in the development process is done accurately. Validation refers to the process of determining if the software, upon completion, fulfills the requirements.

In modeling and simulation, the definition of verification and validation can be stated slightly differently. Verification deals with building a simulation correctly. The accuracy in each step of transforming a model description into a computer simulation is evaluated during verification. Validation is used to determine if the simulation execution, within its domain of applicability, is a sufficiently adequate representation of the entity or system being modeled [7, 8].

V&V of composition-based simulations requires further study. For example, while verification can be performed in the traditional manner for individual components, the verification of a simulation comprised of verified components remains an open issue.

The (re)use of components has also raised the issue of *component certification* [9], the certification that an individual software component satisfies one or more specific constraints/characteristics (e.g., quality, stability, fault-tolerance). In a traditional software development environment, the validation of a component would relate to such constraints. Certification is a concern within a reuse context when the original requirements may not address issues of importance to the component user or may not be readily "visible". Presently, initiatives is this area are generally derived from traditional software engineering approaches influenced by object-oriented technologies and require considerable effort on the part of the potential "consumer". Development of approaches for determining the "quality" (in specific dimensions) of components using a set of well-defined analyses, relying upon the broader field of software metrics, is one viable avenue of activity in this technology area.

Patterns

Patterns, often referred to as design patterns within the object-oriented community, have become a popular technique for supporting the reuse of design information. A pattern describes a problem, a solution to the problem, a context within which that solution is valid, and the consequences or tradeoffs for using the pattern [10, 11]. Patterns are developed for problems that occur repeatedly and then are available for use by developers as appropriate for their designs.

Patterns are abstract in that they are not expressed as code written in some object-oriented programming language but are descriptions of classes or objects and how they may be used to solve the problem at hand [10]. They define the structures and relationships among the elements making up the pattern. Because they are abstract, usually textual in form, patterns have to be implemented each time they are used.

Research is being done in formalisms such as pattern languages for representing design patterns. Mechanisms are being studied for capturing design patterns in a methodical and tangible manner to enhance and facilitate their development and reuse.

Frameworks

A *framework* is an object-oriented reuse technique that can be used to represent a significant subset of an application. Frameworks are less abstract than patterns in that they actually contain code. Patterns are often considered the building blocks of frameworks as frameworks may consist of implementations of one or more patterns [10]. The concept of abstract classes is key to the use of frameworks as a reuse technique. An abstract class is a class with no instances and is used as a template for creating subclasses. The abstract class specifies the interface to and, usually, part of the implementation of its subclasses.

Frameworks are generally domain specific, capturing design decisions specific to that domain thus emphasizing design reuse. This type of reuse leads to a control inversion. When using a library or toolkit, a developer writes the body of a program that calls the code to be reused from the library. When using frameworks, the framework represents the body of the program being reused and the developer, using naming and calling conventions specified by the framework, writes the code to be called. This frees the developer from having to make the design decisions resulting in some loss of creative decisions but providing for faster development of more consistent applications.

When a framework is reused in an application, one of its primary utilities beyond the obvious code reuse is the provision of a broader architecture for the application. Therefore, it is important that a framework be as flexible and extensible as possible to ensure that a large number of applications within the domain can reuse the framework.

Component Repositories

A *component repository* is essentially a library of software components. This library supports the reuse of the components that it contains, providing a variety of services such as component search and retrieval. There are presently examples of large-scale repository structures, generally realized using traditional data base technologies, that seek to support component reuse within limited domain boundaries, e.g., DoD Ada reuse libraries [12]. Due to the challenge presented by effective retrieval as these repositories grow in size, broadly applicable solutions must rely upon technology developments to support search and discovery needs as well as overall performance [13].

Software repository technology may be separated into concerns dealing with the basic elements of component storage and the mechanisms for component distribution or use. The latter, dealing with the access to components in the repository, is intimately coupled to component utilization and such issues as the distributed component model (discussed in a following section).

Component repositories play a central "enabling" role [14] in the effective reuse of components. A critical measure of utility is the effectiveness of the retrieval mechanism. This effectiveness is dependent on many factors, including the representation or characterization of the components in the repository and the architecture of the repository. The former reinforces the importance of an appropriate formalism supporting the required level of retrieval accuracy. The latter is reflected in the repository response time and, more broadly, the overall component architecture (e.g., static components paired with a remote invocation mechanism). These factors are exacerbated in an operational environment in which repositories will in all likelihood be geographical distributed, components are diverse in functional characteristics, and expected end-uses vary widely. The advent of large-scale Internet connectivity and the resulting potential access to massive amounts of information has spurred interest in the development of large, geographically distributed digital libraries [15]. These efforts will continue to address numerous technical issues pertinent to large-scale component storage and repository access. Another needed feature missing from most existing digital libraries is a change management capability to manage the collection of information. Software repositories clearly require similar support given the volatile nature inherent in software.

Search/Discovery technologies support the retrieval of a software component stored in a repository structure. Performance of retrieval algorithms is strongly related to the component (repository) representation. Current practice relies upon traditional data base approaches moderated by object-oriented (i.e., code-level) influences. Necessary research directions include the development of sophisticated meta representations and search mechanisms based on these representations (e.g., [16]). The general goal of these efforts is the improvement of the precision and recall of the search. The recognized need for imprecise queries has led to the development of various technical approaches to support this functionality (e.g., [17, 18]).

Distributed Component Model

The distributed component model is an extension of the component model in which components may reside on different physical hosts. There is considerable work in this area developing such systems with three examples being Microsoft DCOM, OMG CORBA and Java RMI. Currently, most of the systems are static, that is, the components are loaded onto specific host computers, communication links established and there are no architectural changes during software execution [19, 20]. There is ongoing research into operational features such as configuration, administration, and monitoring. Dynamic component migration during execution is also being investigated.

Summary of Findings

The evolution of object technologies into the rapidly expanding area of component software engineering fundamentally supports the composition approach to simulation construction. This evolution continues to yield substantial improvement in technologies applicable to compositional development of software and, specifically, simulations. Thus, the state of *Component Definition* (refer to Figure 2) has been significantly enhanced by the continuing development of programming language mechanisms exemplified by Java Beans and related technologies. These technologies explicitly address many of the language-level requirements for the description and use of individual software components. There is an emerging body of applicable standards in this area that serves to enhance the general usability of technology products [21].

Similarly, *Component Management* issues such as frameworks and component repository approaches have been the focus of very widespread R&D activities. These efforts have yielded products, e.g., the Java Abstract Window Toolkit (AWT) and comprehensive object brokering systems, that have significantly broaden the interest of the technical community in software composition. The area of *Component Utilization* has benefited from large-scale commercial interest in the development of component interconnection. The commercial realizations of *Distributed Component Models*, including CORBA, Java RMI and Microsoft DCOM, have altered the software development landscape and provided mechanisms for effective employment of component architectures in a variety of application domains.

Within the broad context of enabling software composition to be applied to the construction of computer simulations, there are a number of areas requiring additional R&D effort. The current view of composition focuses on components at a largely syntactic level. Although substantial improvements have been realized in the specification of the interface and interconnection syntax of software components, there are very limited mechanisms to describe or use component semantics. Component-based composition allows a user to construct a software system from pre-defined elements (components). These abstract building blocks have an associated semantics that is generally implicitly described at best. Thus, the validity of a composed system often reflects (and requires) the detailed knowledge of the user. Substantial R&D effort will be required to develop technologies supporting software composition in a more general setting.

The description of language semantics has traditionally been a challenge and remains a focus of advanced R&D efforts. As a result, software developers presently have very limited technology products in this area. The ability to define components at the semantic level would have far reaching impact in component creation, management, and utilization. Specifically, explicit representation of component semantics would facilitate component verification & validation, sophisticated repository search and discovery, and verification & validation of composed systems. Similarly, the ability to associate constraints related to composition with individual components would provide a mechanism to express "necessary conditions" for compositional validity. These constraints, as generally realized, define conditions evaluated at the time of composition that characterize required relationships. Though efforts employing a variety of formalisms are underway in this area (e.g., [5, 6]), a comprehensive treatment is still not generally available.

The concept of the component repository is a central feature of most compositional approaches to software construction. Though much of the early work in this area has been in the context of traditional, limited scope software libraries, the current focus of software component technology encompasses additional requirements. These requirements are derived from the desired wider scope of application, spanning larger organizational and geographical boundaries, and place a premium on advanced search mechanisms and effective distributed architectures. In addition, practical requirements in the area of maintenance, long the bane of the software R&D community, are more pressing and challenging in a component-based software development context. Technical solutions that satisfy these requirements are still pending and the success of these solutions will determine in large part how widely applied compositional strategies are in the larger software development community.

In summary, Compositional approaches to the development of large-scale software systems provide a strong foundation for the reuse of software. The last decade has witnessed substantial progress in the definition and successful application of these approaches in the software development and simulation development communities. This paper has presented a technology roadmap depicting the technology areas related to composition. The goal was to review technologies in the broad computing context, though the end-use application area was computer simulations.

References

[1] E. H. Page and J. M. Opper, "Observations on the Complexity of Composable Simulation," presented at 1999 Winter Simulation Conference (submitted to), Phoenix, AZ, 1999.

[2] G. Bundy, S. Slosser, and P. Delaney, "Use Case Definitions for Simulation Composition," 1999.

[3] A. Coen-Porisini, C. Ghezzi, and R. Kemmerer, "Specification of Realtime Systems Using ASTRAL," *IEEE Transactions on Software Engineering*, vol. 23, no. 9, pp. 572-598, 1997.

[4] P. Zave and M. Jackson, "Where Do Operations Come From? A Multiparadigm Specification Technique," *IEEE Transactions on Software Engineering*, vol. 22, no. 7, pp. 508-528, 1996.

[5] M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young, and G. Zelesnik, "Abstractions for Software Architecture and Tools to Support them," *IEEE Transactions on Software Engineering*, vol. 21, no. 4, pp. 314-335, 1995.

[6] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann, "Specification and Analysis of System Architecture Using Rapide," *IEEE Transactions on Software Engineering*, vol. 21, no. 4, pp. 336-355, 1995.

[7] W. O. Balci, "Verification, Validation and Accreditation of Simulation Models," presented at Winter Simulation Conference, 1997.

[8] R. Sargent, "Simulation Model Verification and Validation," presented at 1991 Winter Simulation Conference, 1991.

[9] J. M. Voas, "Certifying Off-the-Shelf Software Components," *IEEE Computer*, vol. 31, no. 6, pp. 53-59, 1998.

[10] R. Johnson, "Frameworks = Components + Patterns," *Communications of the ACM*, vol. 40, no. 10, pp. 10-17, 1997.

[11] H. Gomaa and G. A. Farrukh, "Compositon of Software Architectures from reusable Architecture Patterns," *Proceedings of the Third International Workshop on Software Architecture*, no. , pp. 45-48, 1998.

[12] N.-Y. Lee and C. Litecky, "An Empirical Study of Software Reuse with Special Attention to Ada," *IEEE Transactions on Software Engineering*, vol. 23, no. 9, pp. 537-549, 1997.

[13] R. Seacord, S. Hissam, and K. Mallnau, "Agora: A Search Engine for Software Components," *IEEE Internet Computing*, vol. 2, no. 6, pp. 62-70, 1998.

[14] S. Henninger, "An Evolutionary Approach to Constructing Effective Software Reuse Repositories," *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 2, pp. 111-140, 1997.

[15] B. Schatz, W. Mischo, T. Cole, J. Hardin, A. Bishop, and H. Chen, "Federating Diverse Collections of Scientific Literature," *Computer*, vol. 29, no. 5, pp. 28-36, 1996.

[16] R. Mili, A. Mili, and R. Mittermeir, "Storing and Retrieving Software Components: A Refinement Based System," *IEEE Transactions of Software Engineering*, vol. 23, no. 7, pp. 445-460, 1997.

[17] E. Damiani, M. G. Fugini, and C. Bellettini, "A Hierarchy-Aware Approach to Faceted Classification of Object-Oriented Components," *ACM Transactions on Software Engineering and Methodology*, vol. 8, no. 3, pp. 215-262, 1999.

[18] E. Damiani and M. G. Fugini, "Automatic Thesaurus Construction Supporting Fuzzy Retrieval of Reusable Components," presented at *Proceedings of the 1995 ACM symposium on Applied computing*, 1995.

[19] A. Dogac, C. Dengi, and M. T. Oszu, "Distributed Object Computing Platforms," *Communications of the ACM*, vol. 41, no. 9, pp. 95-103, 1998.

[20] D. Schmidt and M. Fayad, "Lessions Learned Building Reusable OO Frameworks for Distributed Software," *Communications of the ACM*, vol. 40, no. 10, pp. 85-87, 1997.

[21] R. Adler, "Emerging Standards for Component Software," *IEEE Computer*, vol. 28, no. 3, pp. 68-77, 1995.

Appendix - DoD Software Reuse Initiative Technology Roadmap

Reference: DoD Software Reuse Initiative Technology Roadmap (Version 2.2, 30 March 1995).

The following section is a synopsis of the information on composition contained in the DoD SRI Technology Roadmap report. This document is a technology roadmap focused on the broad issue of software reuse. Its discussion of composition characterizes the relevant technologies using the following structure: asset creation; asset management; and asset utilization. Composition is discussed as one of the enabling technologies required for software reuse. Composition is a technology that can be used to develop new programs from existing software building blocks. These building blocks largely retain their identity in the new program.

There are three approaches to composition:

- The first approach relies upon "mining" or "scavenging" existing code for reusable assets or components.
- The second approach reflects the position that reusable assets should be designed, developed and sustained for reuse.
- The third approach is "software schemas" which emphasizes reusable algorithms and data structures rather than source code components.

Mining or scavenging for simple, self-contained pieces of code, such as mathematical routines, was described as being common. Trying to identify more complex building blocks in existing codes is more difficult, often requiring reverse engineering. The sheer number of legacy codes, however, is a compelling reason for continued research in this area and the paper stated that some research was being done.
Components designed and developed for reuse was seen as the current focus of many software engineers. Parameterized programming and inheritance provide support for evolving such reusable assets.
This second approach benefits from classification and library management systems that are being increasingly used as reuse repositories to store and manage collections of reusable assets. Standard bindings, such as SQL and POSIX, also facilitate composition. Programming languages that support the three features of program modularity, polymorphism and inheritance allow programmers to develop software with the properties necessary for reusability, portability and maintainability. Most programming language that support object-oriented programming support these features and, therefore, can be effectively used in the development of reusable components. Object-oriented programming is not the only programming paradigm that could be used to develop reusable software building blocks but is the one currently experiencing rapid growth and

interest.

The third approach to composition, software schemas, is a higher level of abstraction. This technology, still primarily in research stages, is based on formal semantic descriptions of algorithms and data structures. Schema-based transformational languages are needed to transform the formal semantics of schemas into applications.

A reuse repository is described as a database for storing reusable assets while a reuse library is a repository plus an interface for searching, indexing, change management and quality assessment. Reuse libraries have been developed by government agencies including the DoD and NASA. These libraries have addressed many problems related to database security and quality assessment. Several reuse library tools, providing more direct support for reuse, have also been developed commercially and by government agencies. A related, emerging technology is the use of large-scale, internet-based libraries. There are active research activities relating to libraries and information repositories. Indexing methods, interoperability among libraries, performance issues related to obtaining assets, certification of potential library components and efficient presentation of information to users are some of them.

Bibliography

[1] R. Adler, “Emerging Standards for Component Software,” *IEEE Computer*, vol. 28, no. 3, pp. 68-77, 1995.

[2] E. Agerbo and A. Cornils, “How to preserve the benefits of Design Patterns,” presented at Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), 1998.

[3] C. Alexander, “The Origins of Pattern Theory: The Future of the Theory and the Generation of a Living World,” *IEEE Software*, vol. 16, no. 5, pp. 71-82, 1999.

[4] M. Astley and G. Agha, “Customization and Compostion of Distributed Objects: Middleware Abstraction for Policy Management,” presented at Sixth International Symposium on Foundations of Software Engineering, 1998.

[5] O. Astrachan, G. Berry, L. Cox, and G. Mitchener, “Design Patterns: An Essential Component of CS Curricula,” presented at Twenty-Ninth SIGSCE Technical Symposium on Computer Science Education, 1998.

[6] W. O. Balci, “Verification, Validation and Accreditation of Simulation Models,” presented at Winter Simulation Conference, 1997.

[7] A. Bartoli, P. Corsini, G. Dini, and C. Prete, “Graphical Design of Distributed Applications Through Reusable Components,” *IEEE Parallel & Distributed Technology*, vol. 3, no. 1, 1995.

[8] D. Batory and S. O'Malley, “The Design and Implementation of Hierarchical Software Systems with Reusable Components,” *ACM Transactions on Software Engineering and Methodology*, vol. 1, no. 4, pp. 355-398, 1992.

[9] D. Batory and B. Geraci, “Composition Validation and Subjectivity in GenVoca Generators,” *IEEE Transactions on Software Engineering*, vol. 23, no. 2, pp. 67-82, 1997.

[10] I. Ben-Shaul, J. Gish, and W. Robinson, “An Integrated Network Component Architecture,” *IEEE Software*, vol. 15, no. 5, pp. 79-87, 1998.

[11] K. Bobrer, V. Johnson, A. Nilsson, and B. Rubin, “Business Process Components for Distributed Object Applications,” *Communications of the ACM*, vol. 41, no. 6, pp. 43-48, 1998.

[12] A. Borgida and P. Devanbu, “Adding More "DL" to IDL: Toward More Knowledgeable Component Inter-Operability,” presented at International Conference on Software Engineering, 1999.

[13] F. P. Brooks, “No Silver Bullet - Essence and Accidents of Software

Engineering," *IEEE Computer*, vol. 20, no. 4, pp. 10-19, 1987.

[14] A. Brown and K. Wallnau, "The Current State of CBSE," *IEEE Software*, vol. 15, no. 5, pp. 37-46, 1998.

[15] G. Bundy, S. Slosser, and P. Delaney, "Use Case Definitions for Simulation Composition," 1999.

[16] W. Buntine, "Will Domain-Specific Code Synthesis Become a Silver Bullet," *IEEE Intelligent Systems*, vol. 13, no. 2, pp. 9-15, 1998.

[17] G. Campbell, "Adaptable Components," presented at Object-Oriented Programming Systems, Languages and Applications (OOPSLA), Santa Barbara, CA, 1999.

[18] C. Chambers, "Towards Reusable, Extensible Components," *ACM Computing Surveys*, vol. 38, no. 4, Article 192, 1996.

[19] A. Chavez, C. Tornabene, and G. Wiederhold, "Software Component Licensing: A Primer," *IEEE Software*, vol. 15, no. 5, pp. 47-53, 1998.

[20] B. Cheng and J. Jeng, "Reusing Analogous Components," *IEEE Transactions on Knowledge and Data engineering*, vol. 9, no. 2, pp. 341-349, 1997.

[21] A. Coen-Porisini, C. Ghezzi, and R. Kemmerer, "Specification of Realtime Systems Using ASTRAL," *IEEE Transactions on Software Engineering*, vol. 23, no. 9, pp. 572-598, 1997.

[22] J. Cook and J. Dage, "Highly Reliable Upgrading of Components," presented at International Conference on Software Engineering, 1999.

[23] J. O. Coplien, "Reevaluating the Architectural Metaphor: Toward Piecemeal Growth," *IEEE Software*, vol. 16, no. 5, pp. 40-44, 1999.

[24] D. Corkill, "Countdown to Success: Dynamic Objects GBB and Radarsat-1," *Communications of the ACM*, vol. 40, no. 5, pp. 48-58, 1997.

[25] E. Damiani and M. G. Fugini, "Automatic Thesaurus Construction Supporting Fuzzy Retrieval of Reusable Components," presented at *Proceedings of the 1995 ACM symposium on Applied computing*, 1995.

[26] E. Damiani, M. G. Fugini, and C. Bellettini, "A Hierarchy-Aware Approach to Faceted Classification of Object-Oriented Components," *ACM Transactions on Software Engineering and Methodology*, vol. 8, no. 3, pp. 215-262, 1999.

[27] F. DeRemer and H. H. Kron, "Programming-in-the-large versus programming-in-the-small," *IEEE Transactions on Software Engineering*, vol. 2,

no. 6, pp. 80-86, 1976.

[28] T. Digre, “Business Object Component Architecture,” *IEEE Software*, vol. 15, no. 5, pp. 60-69, 1998.

[29] A. Dogac, C. Dengi, and M. T. Oszu, “Distributed Object Computing Platforms,” *Communications of the ACM*, vol. 41, no. 9, pp. 95-103, 1998.

[30] S. Dumas and G. Gardain, “A Workbench for Predicting the Performances of Distributed Object Architectures,” *1998 Conference on Winter Simulation*, no. , pp. 515-522, 1998.

[31] M. Fayad and D. Schmidt, “Object-Oriented Application Frameworks,” *Communications of the ACM*, vol. 40, no. 10, pp. 32-38, 1997.

[32] E. Fernandez, “Building Systems Using Analysis Pattern,” presented at Third International Workshop on Software Architecture, 1998.

[33] W. B. Frakes and C. J. Fox, “Quality Improvement Using A Software Reuse Failure Modes Model,” *IEEE Transactions on Software Engineering*, vol. 22, no. 4, pp. 274-279, 1996.

[34] M. Franz, “Dynamic Linking of Software Components,” *IEEE Computer*, vol. 30, no. 3, pp. 74-81, 1997.

[35] G. Froehilich, J. J. Hoover, L. Liu, and P. Sorenson, “Hooking into Object-Oriented Application Frameworks,” *International Conference on Software Engineering*, no. , pp. 491-501, 1997.

[36] D. Gannon, R. Bramley, T. Stuckey, J. Villacis, J. Balasubramanian, E. Akman, F. Breg, S. Diwan, and M. Govindaraju, “Developing Component Architectures for Distributed Scientific Problem Solving,” *IEEE Computational Science and Engineering*, vol. 5, no. 2, 1998.

[37] D. Garlan and M. Shaw, “An Introduction to Software Architecture,” Carnegie Mellon University (School of Computer Science), Pittburgh, PA 15213-3890 CMU-CS-94-166, January 1994.

[38] D. Garlan, G. E. Kaiser, and D. Notkin, “Using tool abstraction to compose systems,” *IEEE Computer*, vol. 25, no. 6, 1992.

[39] D. Garlan, R. Allen, and J. Ockerbloom, “Exploiting Style in Architectural Design Environment,” *Proc. SIGSOFT'94: Foundations of Software Engineering*, no. , pp. 175-188, 1994.

[40] D. Garlan, R. Allen, and J. Ockerbloom, “Architectural Mismatch or Why it's hard to build Systems out of Existing Parts,” presented at 17 International Conference on Software Engineering, Los Alamitos, CA, 1995.

[41] H. Gomaa and G. A. Farrukh, "Compositon of Software Architectures from reusable Architecture Patterns," *Proceedings of the Third International Workshop on Software Architecture*, no. , pp. 45-48, 1998.

[42] D. Gray, J. Hotchkiss, S. LaForge, S. Shalit, and T. Winbery, "Modern Languages and Microsoft's Components Object Model," *Communications of the ACM*, vol. 41, no. 5, pp. 55-65, 1998.

[43] T. Hansen, "Development of Successful Object-Oriented Frameworks," presented at Addendum to the 1997 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 1997.

[44] S. Heiler, R. Miller, and V. Ventrone, "Using Metadata to Address Problems of Semantic Interoperability in Large Object Systems," presented at First IEEE Metadata Conference, Silver Spring, MD, 1996.

[45] J. A. Heim, "Integrating Distributed Simulation Objects," presented at Proceedings of the 1997 Winter Simulation Conference, 1997.

[46] S. Henninger, "An Evolutionary Approach to Constructing Effective Software Reuse Repositories," *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 2, pp. 111-140, 1997.

[47] J.-M. Jezequel and B. Meyer, "Design by Contract: The Lessons of Ariane," *IEE Computer*, vol. 30, no. 1, pp. 129-130, 1997.

[48] R. Johnson, "Frameworks = Components + Patterns," *Communications of the ACM*, vol. 40, no. 10, pp. 10-17, 1997.

[49] R. Keller and R. Schauer, "Design Components: Towards Software Composition at the Design Level," presented at 20th International Conference on Software Engineering, Kyoto, Japan, 1998.

[50] N. L. Kerth and W. Cunningham, "Using Patterns to Improve Our Architectural Vision," *IEEE Software*, vol. 14, no. 1, pp. 53-59, 1997.

[51] L. Kortright, "An Incremental Approach to the Development of Reusable General-Purpose Discrete-Event Simulator Components," presented at Proceedings of the 1994 Conference on TRI-Ada, 1994.

[52] J. Kotula, "Using Patterns to Create Component Documentation," *IEEE Software*, vol. 15, no. 2, 1998.

[53] D. Krieger and R. Adler, "The Emergence of Distributed Component Platforms," *IEEE Computer*, vol. 31, no. 3, pp. 43-53, 1998.

[54] D. Krieger and R. M. Adler, "The Emergence of Distributed Component

Platforms," *IEEE Computer*, vol. 31, no. 3, pp. 43-53, 1998.

[55] K. Kroeker, "Software [R]evolution: A Roundtable," *IEEE Computer*, vol. 32, no. 5, pp. 48-57, 1999.

[56] N.-Y. Lee and C. Litecky, "An Empirical Study of Software Reuse with Special Attention to Ada," *IEEE Transactions on Software Engineering*, vol. 23, no. 9, pp. 537-549, 1997.

[57] S. M. Lewandowski, "Framework for Component-Based Client/Server Computing," *ACM Computing Surveys*, vol. 30, no. 1, pp. 3-27, 1998.

[58] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann, "Specification and Analysis of System Architecture Using Rapide," *IEEE Transactions on Software Engineering*, vol. 21, no. 4, pp. 336-355, 1995.

[59] M. D. McIlroy, "Mass Produced Software Components," presented at Nato Conference on Software Engineering, 1968.

[60] J. Meekel, T. Horton, r. France, C. Mellone, and S. Dalvi, "From Domain Models to Architecture Frameworks," presented at 1997 Symposium on Software Reusability, 1997.

[61] M. Mezini and K. Lieberherr, "Adaptive Plug-and-Play Components for Evolutionary Software Development," presented at Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 1998.

[62] H. Mili, F. Mili, and A. Mili, "Reusing Software: Issues and Research Directions," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, 1995.

[63] H. Mili, H. Sahraoui, and I. Benyahia, "Representing and querying Reusable Object Frameworks," presented at 1997 Symposium on Software Reusability, 1997.

[64] R. Mili, A. Mili, and R. Mittermeir, "Storing and Retrieving Software Components: A Refinement Based System," *IEEE Transactions of Software Engineering*, vol. 23, no. 7, pp. 445-460, 1997.

[65] R. T. Monroe, A. Kompanek, R. Melton, and D. Garlan, "Architectural Styles, Design Patterns, and Objects," *IEEE Software*, vol. 14, no. 1, pp. 43-52, 1997.

[66] M. Morgenstern, "Metadata for Heterogeneous Databases," presented at IEEE Second Metadata Conference, 1997.

[67] M. Nanard, J. Nanard, and P. Kahn, "Pushing Reuse in Hypermedia Design: Golden Rules, Design Patterns and Constructive Templates," presented

at Ninth ACM Conference on Hypertext and Hypermedia, 1998.

[68] J. Ning, "An Architecture Design Environment for Component-Based Software Engineering," *Proceedings of the 1997 International Conference on Software Engineering (ICSE'97)*, no. , pp. 614-615, 1997.

[69] E. H. Page and J. M. Opper, "Observations on the Complexity of Composable Simulation," presented at 1999 Winter Simulation Conference (submitted to), Phoenix, AZ, 1999.

[70] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053-1058, 1972.

[71] L. Perrochon and W. Mann, "Inferred Designs," *IEEE Software*, vol. 16, no. 5, pp. 46-51, 1999.

[72] S. L. Pfleeger, R. Jeffery, B. Curtis, and B. Kitchenham, "Status Report on Software Measurement," *IEEE Software*, vol. 14, no. 2, pp. 33-43, 1997.

[73] R. Rasala, "Function Objects, Function Templates, and Passage by Behavior in C++," *Proceedings of the Twenty-Eight SIGCSE Technical Symposium on Computer Science Education*, no. , pp. 35-38, 1997.

[74] J. D. Riley, "A Comparison of Two Approaches to Distributed Application Development with Ada: The Ada Distributed Systems Annex and CORBA," *ACM 0-89791-808-8/96/0012 3.50*, no. , pp. 73-80, 1996.

[75] D. Rine, N. Nada, and K. Jaber, "Using Adapters to Reduce Interacting Complexity in Reusable Component-Based Software Development," presented at Proceedings of the Fifth Symposium on Software Reusability, 1999.

[76] R. Sargent, "Simulation Model Verification and Validation," presented at 1991 Winter Simulation Conference, 1991.

[77] B. Schatz, W. Mischo, T. Cole, J. Hardin, A. Bishop, and H. Chen, "Federating Diverse Collections of Scientific Literature," *Computer*, vol. 29, no. 5, pp. 28-36, 1996.

[78] H. A. Schmid, "Creating Applications from Components: A Manufacturing Framework Design," *IEEE Software*, vol. 13, no. 6, pp. 67-75, 1996.

[79] D. Schmidt, "Using Design Patterns to Guide the Development of Reuseable Object-Oriented Software," *Computing Surveys*, vol. 28, no. 4es, Article 162, 1996.

[80] D. Schmidt and M. Fayad, "Lessions Learned Building Reusable OO Frameworks for Distributed Software," *Communications of the ACM*, vol. 40, no. 10, pp. 85-87, 1997.

[81] R. Seacord, S. Hissam, and K. Mallnau, "Agora: A Search Engine for Software Components," *IEEE Internet Computing*, vol. 2, no. 6, pp. 62-70, 1998.

[82] M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young, and G. Zelesnik, "Abstractions for Software Architecture and Tools to Support them," *IEEE Transactions on Software Engineering*, vol. 21, no. 4, pp. 314-335, 1995.

[83] R. Smith, A. Parrish, and J. Hale, "Cost estimation for Component Based Software Development," presented at 36th Annual Southeast Regional Conference, 1998.

[84] S. Srinivasan and J. Vergo, "Object Oriented Reuse: Experience in Developing a Framework for Speech Recognition Applications," presented at Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), 1998.

[85] P. Steyaert, c. Lucas, K. Mens, and T. D'Hondt, "Managing the Evolution of Reusable Assets," presented at Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), 1996.

[86] J. Sutherland, "Why I Love the OMG: Emergence of a Business Object Componet Architecture," *StandardView*, vol. 6, no. 1, pp. 4-13, 1998.

[87] R. N. Taylor, N. Medividovic, K. M. Anderson, E. J. W. Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrown, "A Component and Message-Based Architectural Style for GUI Software," *IEEE Transcations on Software Engineering*, vol. 22, no. 6, pp. 390-406, 1996.

[88] W. Tracz, "Developing Reusable Java Components," *Proceedings of the Third International Workshop on Software Architecture*, no. , 1997.

[89] J. Voas, "Maintaining Component-Based Systems," *IEEE Software*, vol. 15, no. 4, 1998.

[90] J. M. Voas, "Certifying Off-the-Shelf Software Components," *IEEE Computer*, vol. 31, no. 6, pp. 53-59, 1998.

[91] Y. M. Wang and P. Y. E. Chung, "Customization of Distributed Systems Using COM," *IEEE Concurrency*, vol. 6, no. 3, pp. 8-12, 1998.

[92] E. J. Weyuker, "Testing Component-Based Software: A Cautionary Tale," *IEEE Software*, vol. 15, no. 5, pp. 54-59, 1998.

[93] M. Wooldride and N. Jennings, "Pitfalls fo Agent-Oriented Development," presented at Second International Conference on Autonomous Agents, 1998.

[94] D. Wu, D. Agrawal, and A. E. Abbadi, "StratOSphere: Mobile Processing of Distriubted Objects in Java," presented at The Fourth Annual ACM/IEEE

International Conference on Mobile Computing and Networking, Dallas, Texas, 1998.

[95] P. Zave and M. Jackson, “Where Do Operations Come From? A Multiparadigm Specification Technique,” *IEEE Transactions on Software Engineering*, vol. 22, no. 7, pp. 508-528, 1996.

[96] S. Zweben, S. Edwards, B. Weide, and J. Hollingsworth, “The Effects of Layering and Encapsulation on Software Development Cost and Quality,” *IEEE Transactions on Software Engineering*, vol. 21, no. 3, pp. 200-208, 1995.