Software Reuse Research: Status and Future

William B. Frakes and Kyo Kang

Abstract—This paper briefly summarizes software reuse research, discusses major research contributions and unsolved problems,provides pointers to key publications, and introduces four papers selected from The Eighth International Conference on Software Reuse (ICSR8).

Index Terms—Software reuse, domain engineering, research, metrics, architectures, generators, finance.

INTRODUCTION

THIS paper briefly summarizes software reuse research, discusses major research contributions and unsolved problems, provides pointers to key publications, and introduces four papers from The Eighth International Conference on Software Reuse (ICSR8) selected on the recommendations of conference reviewers and attendees.

We have been helped in writing this paper by responses to a brief survey of longtime reuse researchers and practitioners who were asked four questions:

What are the top three contributions from reuseresearch?

What are the top three remaining problems for reuse research?

What are the top three references in your area of reuse research?

Anything else you would suggest for inclusion? Their responses varied widely, both by topic and what they considered important. Despite this variability certain

common themes emerged, and we will discuss these in

greater detail.

We begin with some basic definitions. Software reuse is the use of existing software or software knowledge to construct new software. *Reusable assets* can be either reusable software or software knowledge. *Reusability* is a property of a software asset that indicates its probability ofreuse.

Software reuse's purpose is to improve software quality and productivity. Reusability is one of the "illities" or major software quality factors. Software reuse is of interest because people want to build systems that are bigger and more complex, more reliable, less expensive and that are delivered on time. They have found traditional software engineering methods inadequate, and feel that software reuse can provide a better way of doing software engineering.

A key idea in software reuse is domain engineering (aka product line engineering). The basic insight is that most

software systems are not new. Rather they are variants of systems that have already been built. Most organizations build software systems within a few business lines, called domains, repeatedly building system variants within those domains. This insight can be leveraged to improve the quality and productivity of the software production process.

HISTORY

Software reuse has been practiced since programming began. Reuse as a distinct field of study in software engineering, however, is often traced to Doug Mcilroy's paper which proposed basing the software industry on reusable components. Other significant early reuse research developments include Parnas' idea of program families and Neighbors' introduction of the concepts of domain anddomain analysis.

Active areas of reuse research in the past twenty years include reuse libraries, domain engineering methods and tools, reuse design, design patterns, domain specific soft- ware architecture, componentry, generators, measurement and experimentation, and business and finance. Important ideas emerging from this period include systematic reuse, reuse design principles such as the three C's model, module interconnection languages, commonality/variability analy- sis, variation point, and various approaches to domain specific generators.

While these areas comprise the core of reuse research, software reuse research and practice has deep and complex interactions with other areas of computer science and software engineering. For example, though its developers did not consider themselves as doing reuse research per se, reuse was clearly a key design goal of the Unix program- ming environment. The C language was designed to besmall and augmented with standard libraries of reusable functions. Shell programming languages are based on reusable filter programs that are combined via a module interconnection language—data pipes. The C++ language was also designed to encourage reuse as described in [1].

Other areas of computer science research of key relevance to reuse are abstract datatypes and object oriented methods, programming language theory, software architec- tures, compilers, software development process models, metrics and experimentation, and organizational theory.

BUSINESS AND FINANCE

The ultimate purpose of domain engineering and systema-tic software reuse is to improve the quality of the products and services that a company provides and, thereby, maximize profits. It is easy to lose sight of this goal when considering the technical challenges of software reuse and yet, software reuse will only succeed if it makes good business sense. Capital can be expended by an organization in many ways to maximize return to shareholders. Software reuse will only be chosen if a good case can be made that it is the best alternative choice for use of capital.

Business related reuse research has identified organiza-

tional structure to support corporate reuse programs, staged process models for reuse adoption, and models for estimat- ing return on investment from a reuse program. More recent work has extended the return on investment analysis to include benefits from strategic market position [2].

Important problems remaining in this area include:

- Sustaining reuse programs.
- Tech transfer.
- Reuse and corporate strategy.
- Organizational issues.
- Process focus.

We will now discuss some of these issues.

3.1 Process Focus

Implementing a reuse program in a corporate environmentrequires a decision about when and where a capital investment is to be made. Development of reusable assets often requires a capital investment and there should be a strategic decision as to whether investment will be made proactively or reactively.

Proactive investment for software reuse is like the waterfall approach in conventional software engineering. The target domain or product line is analyzed, architectures for the domain are defined, then reusable assets are designed and implemented taking foreseeable product variations into account. This approach tends to require a large upfront investment, and returns on investment can only be seen when products are developed and maintained. This approach might be suited to organizations that can predict their product line requirements well into the future and that have the time and resources for a long develop- ment cycle. There is an investment risk with this approach if product line requirements deviate from the projections. The cost for evolving reusable assets and retrofitting products with new assets can be high.

Reactive investment is an incremental approach to asset building. One develops reusable assets as reuse opportu- nities arise while developing products. A subdomain with a clear problem boundary and projected requirements varia- tions might be a good candidate. This approach is advantageous in that the asset development costs can be distributed over several products and no upfront large capital investment is necessary. However, if there is no sound architectural basis for the products in a domain, this approach can be costly as existing products may continu- ously have to be reengineered when assets are developed.

This approach works in situations where the requirements for product variations cannot be predicted well in advance. Another approach proposed by Charles Kruger, called the extractive model, stays in between the proactive and reactive approaches [3]. The extractive approach reuses one or more existing software products for the product line's initial baseline. This approach can be effective for an organization that has accumulated development experi- ences and artifacts in a domain but wants to quickly transition from conventional to software product line engineering. When accumulated expertise is used properly,

this approach may not require a large capital investment.

Organizational Issues

There are two types of commonly observed organizational approaches to establishing a reuse program: centralized and distributed asset development.

The centralized approach typically has an organizational unit dedicated to developing, distributing, maintaining, and, often, providing training about reusable assets. The unit has responsibilities to analyze commonalities and variabilities of applications within the product line that have been developed or that will be developed in the future. The unit also develops standard architectures and reusable assets, and then makes them available to develop- ment projects. The unit maintains these assets and, often, also supports customization. The cost of this organizational unit is amortized across projects within the product line.

Some of the advantages of this approach are that 1) the product line-wide engineering vision can be shared among the projects easily, 2) development knowledge and corpo- rate expertise can be utilized efficiently across projects, and

3) assets can be managed systematically. There are also disadvantages with this approach. This approach often requires a large upfront capital investment to create an organizational unit dedicated to implementing a reuse program and it takes time to see a return on investment. Also, experts may have to be pulled away from on-going projects to create a centralized core expert group, which may face strong resistance from project managers. There- fore, to be successful, there must be a strong commitment from upper management. Distributed, Collaborative

With the distributed approach, a reuse program is implemented collaboratively by projects in the same product line. Each project has a responsibility to contribute reusable assets to other participating projects and, therefore, asset development and support responsibilities are distrib-uted among projects.

The obvious advantages of this approach are that 1) there is less overhead cost as there is no need to create a separate organizational unit, and 2) asset development costs are distributed among projects. No large up-front investment isnecessary.

Some of the disadvantages are that 1) it may be difficult to coordinate asset development responsibilities if there isno common vision for the reuse program; 2) even if there is a shared vision among projects, it may not be easy for a given project to provide a component that meets the needs of other projects; and 3) there must be a convincing cost/benefit

model to solicit active participation. There is a danger that projects may be willing to use other's products, but will be reluctant to make investments for others.

MEASUREMENT AND EXPERIMENTATION

If software reuse is to be based on science and engineering, it must be treated as an empirical discipline. The develop- ment of concepts such as reuse and reusability has naturally led to questions of how to measure them, and of how to run experiments to establish their impact on quality and productivity. Metrics have been defined for many areas of software reuse [4]. These include classification models of types of reuse, reuse library metrics, cost benefit models, maturity assessment models, amount of reuse metrics, failure modes models, and reusability assessment models. A software metric is a quantifiable measurement of an attribute of a software product or process. A model is a stated relationship among metric variables. Experimenta- tion is the process of

establishing the effect of one set of variables on another.

Experiments in software reuse have included studies of indexing methods for reusable components and correla- tional studies of the relationship between reuse, quality, and productivity. Much data on the effect of reuse on important variables such as cost of software production, time to market, and project completion time has also been reported, though these studies tend to be quasi-experi- mental. Such studies have the typical problems of field studies in trying to control the internal validity of the experiments. Measurement and experimentation of reuse and domain engineering is one area where much more work is needed.

COMPONENTRY

The broad interest in component-based software engineer- ing has resulted in several component development, integration and deployment technologies. Most noted of these are Object Management Group (OMG)'s Common Object Request Broker Architecture (CORBA) Component Model (CCM), Sun's Enterprise JavaBeans (EJB), and Microsoft's Component Object Model (COM+).

CORBA CCM allows integration and invocation of distributed components without concern for object location, programming language, operating system, communication protocol, or hardware platform. Concerns that cut across components, such as transaction handling, security, persis- tent state management, and event notification, are sup- ported by CORBA Object Services (COS).

EJB along with Java Remote Method Invocation (RMI) provides, as with CORBA, a platform for developing, integrating, and deploying distributed components. EJB provides an environment for handling complex features of distributed components such as transaction management, connection pooling, state management, and multithreading. This technology depends on the Java language but it achieves platform independence through the language. EJB, together with J2EE and Java servlets, provides a middleware platform for developing Web applications.

COM+ provides runtime services, such as transaction management, synchronization, threading, and object pool- ing, for developing distributed applications on Microsoft's Windows platform. While permitting integration of binary components written in any language, COM, that works under COM+, requires them to obey the rules of COM component identity, lifetime and binary layout, and writing the plumbing code to create a COM component. .NET frees one from having to obey all these rules and write extra code and allows development of applications accessing distrib- uted systems on internal corporate networks or the Internet. These technologies are still evolving, but they provide important middleware platforms on which reusable com- ponents can be developed, applications can be created integrating these components, and applications thus created
can run.

Each of the technologies discussed above supports a set of features, or concerns, such as security, that cut across a number of components. Aspect oriented programming supports implementation of these cross-cutting concerns, called aspects, and integration of these into functional components.

DOMAIN ENGINEERING (PRODUCT LINE ENGINEERING)

Technologies for high software productivity through domain engineering started to appear in early 1980s, but, application of these technologies in industrial settings and stories of successes have only been reported recently. One such report is the paper by Van Ommering in this issue. The paper in this issue by Moon et al. discusses an approach to the important problem of handling requirements for systems in a product line.

In this section, we briefly review several domain engineering (aka product line engineering) approaches reported in recent publications. The technologies reviewed in this section fall largely into two categories: process and technique. FAST defines a product line engineering process model. All others are development techniques but these techniques compliment each other in that DARE focuses on extracting information from existing code and documents to help analysts create domain models, FORM focuses on the commonality and variability analysis of the features of applications in a product line to use as a foundation for creating architectures and components, KobrA defines both processes and techniques for developing components and integrating them to create applications, and Koala has a component focus and provides a mechanism for integrating components. PLUS provides UML extensions to support product line engineering. Each of these technologies is summarized below.

DARE

DARE, domain analysis and reuse environment, is a method and toolset for doing domain engineering [5]. One of the major research goals of DARE was to explore how much of domain analysis can be based on a repeatable process and how much can be automated. The DARE process draws on three sources of information: code, documents, and expert knowledge as the basis for domainmodels. Information extracted from these three sources is used to build domain models such as facet tables and templates, feature tables, and generic architectures. All information and models are stored in a domain book. DARE has been used successfully in industry, for example, to support the building of text and database systems at Oracle [6].

6.1 FAST

Lucent Technologies introduced Family-Oriented Abstrac- tion, Specification, and Translation (FAST) method in 1999 [7]. FAST defines a pattern of engineering processes that are commonly used in product line engineering. FAST consists of three subprocesses: domain qualification (DQ), domain engineering (DE), and application engineering (AE). DQ identifies a product line worthy of investment, DE develops product line assets and environments, and AE develops products rapidly by using the product line assets.

FAST focuses on the processes for product line engineer- ing and it has been applied to the product line of telecommunication infrastructure and systems at Lucent Technology.

6.2 FORM

Feature-Oriented Reuse Method (FORM) was developed at Pohang University of Science and Technology (POSTECH)
[8] and is an extension of the Feature-Oriented Domain Analysis (FODA) method [9]. FORM is a systematic method that looks for and captures commonalties and variabilities of a product line in terms of "features." These analysis results are used to develop product line architectures and components. The model that captures the commonalties and variabilities is called a feature model. It is used to support both engineering of reusable

product line assets and development of products using the assets.

This method has been applied to several industrial application domains, including electronic bulletin board systems, PBX, elevator control systems, yard inventory systems, and manufacturing process control systems, to create product line software engineering environments andsoftware assets [10]. FORM includes techniques and tools for product line engineering but has a loose process structure.

6.3 KobrA

Fraunhofer IESE has been developing the KobrA method, a component based product line engineering approach with UML. KobrA is an abbreviation of Komponentbasierte Anwendungsentwicklung and means a component-based application development method [11]. KobrA provides an approach to developing generic assets that can accommo- date variations of a product line through framework engineering. The framework engineering starts with designing a context under which products of a product line will be used. The context includes information on the scope, commonality, and variability of the product line. Then, product line requirements are analyzed and the Komponent (i.e., KobrA component) specifications are developed. Based on the specifications, the Komponent realizations, which describe the design that satisfies the requirements, are

developed. KobrA also provides a decision model that constrains the selection of variations for the valid config- uration of products. KobrA includes both processes and techniques for product line engineering.

6.4 PLUS

Product Line UML-Based Software Engineering (PLUS) extends the UML-based modeling methods for single systems development to support software product lines [12]. PLUS provides various modeling techniques and notations for product line engineering. First, for the software product line requirements engineering activity, use case modeling and feature modeling are provided. Second, for the software product line analysis activity, static modeling, dynamic interaction modeling, dynamic state machine modeling, and feature/class dependency model- ing are introduced. Last, for the software product line design activity, software architecture patterns and component-based software design are proposed. PLUS extends UML by integrating various product line engineering techniques to support UML-based product line engineering.

6.5 Koala

Koala, developed at Philips Corp. for analysis of embedded software in the domain of electrical home appliances, is an architecture description language [13] for product lines. Koala is a descendant of Darwin [14] and is designed based on the experience of applying Darwin to television software systems. In Koala, *diversity* interfaces and *switches* are provided for handling product variations. The diversity interfaces can be used to handle the internal diversity ofcomponents and the switch can be used to route connec- tions between interfaces. When a component provides some extra functions, the access to these functions can be defined as optional interfaces. This enables the optimization of the code at compile time. Koala is a component-based product line engineering method with tools for integrating compo- nents both at compile-time and at runtime.

PROGRAMMING LANGUAGES

The evolution of programming languages is tightly coupled with reuse in two important ways. First, programming languages have evolved to allow developers to use ever larger grained programming constructs, from ones and zeroes to assembly statements, subroutines, modules, classes, frameworks, etc. Second, programming languages have evolved to be closer to human language, more domain focused, and therefore easier to use. Languages such as Visual C++, Delphi, and Visual Basic clearly show the influence of software reuse research. The paper on the Fusion system by Weber et al. in this issue is a continuation of the trend of making large grained domain specific programming constructs, in this case business rules, available in a form closer to the language used by workers in the domain. Fusion also mixes declarative and algorith- mic programming language approaches in a single system.

Systematic reuse via domain engineering is another step in this direction. In systematic reuse, we consider how to

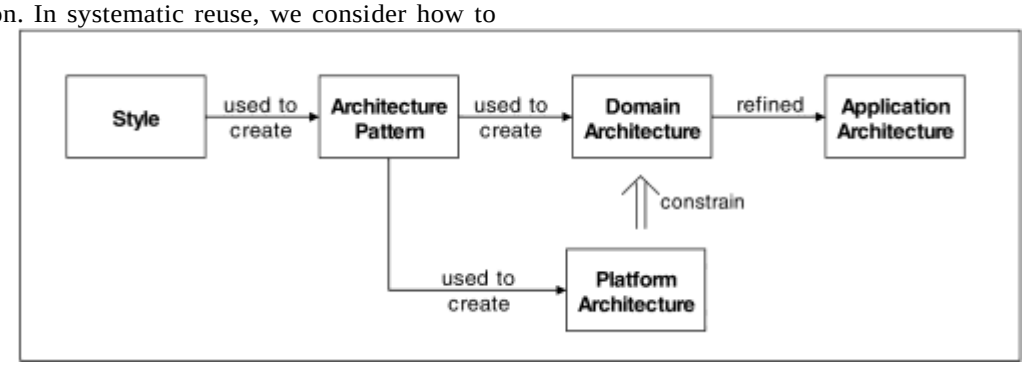

Fig. 1. Architecture Concepts.

codify and reuse subsystems and architectures. We attempt to establish the required vocabulary for a given problem area, apply it to the system building environment for that domain, and, thereby, build higher quality systems more productively.

Reuse research has contributed to the widespread practice of *design to interfaces*, the practice of separating interfaces from implementations, and to the common use

of off the shelf libraries of general components such as those for C, C++, Java, and C#. Some research on restricting theuse of pointers in languages and better ways of handling reference aliasing has also been active.

1 LIBRARIES

A reuse library consists of a repository for storing reusable assets, a search interface that allows users to search for assets in the repository, a representation method for the assets, and facilities for change management and quality assessment. Much research on reuse libraries has been done as reported in the papers in the reuse roadmap. Key ideas are the application of indexing methods such as free text keyword and faceted classification to reusable components. There has been disagreement in the reuse research community about the importance of libraries for reuse. However, failure modes analysis of the reuse process shows that in order to be reused a component must be available, findable, and understandable. A reuse library supports all of these. The argument has also been made that most component collections are small and, therefore, do not need sophisticated library support. However, the emergence of the World Wide Web as a defacto standard library of
reusable assets argues against this point of view.
Experiments on reuse libraries indicate that current methods of component representation could be improved. There is also a need for library environments that include facilities for configuration management and that integrate facilities for measurements such as usage and return on investment. The paper by de Jonge in this issue discusses how to handle the build process for reusable components.

2 ARCHITECTURES

Since the late 1980's software architecture has been recognized as an important consideration for reusing software. Architectural decisions because they occur early
in the software lifecycle, have a strong impact on system quality attributes. Architectural decisions are also difficult to change late in the lifecycle.
Software architecture may be explored at different levels of abstraction. Shaw explored various structural models called architecture styles, that were commonly used in software and then examined quality attributes related to each style. At a lower level of abstraction than style, [15] identified architectural patterns that commonly occur in various design problem domains such as client-server architectures, proxies, etc. In theory, these architecture patterns can be defined by applying a combination of architecture styles.
Using architecture patterns, reference architectures for
an application domain or a product line can be built. These architectures embody application domain-specific seman- tics and quality attributes inherited from the architecture patterns. Application architectures may be created using domain architectures. Examples of domain architectures are reported in [16].
Platform architectures are middleware on/with which
applications and components for implementation of an application can be developed. Examples of these are CORBA, COM+, and J2EE. A platform architecture selected for implementation of applications in a domain may influence architectural decisions for a domain architecture. For example, transaction management is supported by most of platform architectures and a domain architecture may use facilities provided by the platform architecture selected for the domain.
The relationships between these concepts related to
architectures are summarized in Fig. 1.

3 GENERATIVE METHODS

An important approach to reuse and one tightly coupled to the domain engineering process is generative reuse. Gen- erative reuse is done by encoding domain knowledge and relevant system building knowledge into a domain specific application generator. New systems in the domain are created by writing specifications for them in a domain specific specification language. The generator then trans- lates the specification into code for the new system in a target language. The generation process can be completely automated, or may require manual intervention.

Important contributions to generative reuse include the development of the theory of metacompilers, also known as application generator generators. These tools assist in the development of domain specific application generators.

An important part of making domain engineering repeatable is a clear mapping between the outputs of domain analysis and the inputs required to build applica- tion generators. Better integration of these two phases of domain engineering will mean much improved environ-ments for domain engineering.

4 RELIABILITY AND SAFETY

Better system reliability is one of the goals of software reuse. It is argued that reusable components, because of more careful design and testing and broader and more extensive usage, can be more reliable that one use equivalents. If so, then it is further argued that using these more reliable components in a system architecture can increase the reliability of the system as a whole. Higher system reliability via generative reuse is based on the idea that replacing error prone human processes in software development by automation can produce a more reliable system. There are many open research questions in this area that need to be addressed before these hypotheses can be verified.
A topic related to reliability is software safety. Two software safety failures have been attributed to reuse. In the Therac-25 system, a software component carried over from a previous version of the system caused the machine to malfunction resulting in the loss of several lives [17]. In the Ariane project, failure of a software component, caused the loss of a rocket costing around half a billion dollars [18].
Of concern regarding reliability and safety of compo- nents is emergent behavior, defined as system behavior that cannot be predicted on the basis of the behavior of components comprising the system. As components are designed to be more autonomous and intelligent, unpre- dictable system behavior based on component interactions is an area of needed research.

5 FUTURE RESEARCH

Though significant progress has been made on software reuse and domain engineering, many important problems remain. One of these has to do with scalability which is the problem of applying reuse and domain engineering methods to very large systems. One important issue is how to make best use of reusable components for systems of this size. Another is how to do sufficient formal specifications of architectures to support the automated construction of very large systems. Reuse and domain

engineering methodologies also need to be wider spectrum, that is applicable to a broader range of software domains.

Better representation mechanisms for all software assets, including means for specification and verification, are needed. Researchers point to the need for support and enforcement of behavioral contract specifications for components. This may be summarized as a movement from *design to interfaces* to *design by contract*. They also argue for better methods for specification and reasoning support for
popular component libraries, and for work on elimination of reference semantics in industrial languages.

Another important problem is sustainability. There have now been many industrial reuse programs. A current problem is to find the means of sustaining reuse programs on a long-term basis. One approach to this problem will be determining how to make better links between reuse and domain engineering and corporate strategy. Related to this question is identifying what should be made reusable, that is, which reusable corporate products and processes will give the highest return on investment? Another is deter- mining how to do better technology transfer; that is, how to better support practitioners in the application of reuse and domain engineering research. Needed for this is a deeper understanding of when to use particular methods, based, for example, on system size and business context. There is also a need for a seamless integration between the models output from domain analysis and the inputs needed to for domain implementations such as components, domain specific languages, and application generators.

As discussed above, safety and reliability issues are important and must be adequately addressed if reuse is to be a common practice. Another area of potentially interest- ing research concerns the relationship of reuse and domain engineering to newer software development processes such as agile methods.

A key element in the success of reuse and domain engineering is the ability to predict needed variabilities in future assets. This is sometimes called the oracle hypoth- esis. Richer means of specifying potential reuser needs is an area needing research. This will involve a method for clearly stating reuse contexts and assumptions.

There is a clear need for much more empirical work on reuse and domain engineering. Research is needed to identify and validate measures of reusability, including good ways to estimate the number of potential reuses.

Industry studies have shown that education is a primary factor in better reuse, yet there had been little systematic study of how best to do reuse education. Certainly, both academia and industry could improve educational practices. One way to do this and to facilitate better reuse technology transfer would be better joint work between industry and academia.

Currently, most reuse research focuses on creating and integrating adaptable components at development or at compile time. However, with the emergence of ubiquitous computing, reuse technologies that can support adaptation and reconfiguration of architectures and components at runtime are in demand. One implication of this development is that we somehow need to embed engineering know-how into code so it can be applied while an application is running. More research on self-adaptive software, reconfigurable context-sensitive software, and self-healing systems is needed.

Reuse research has been ongoing since the late 1960s and domain engineering research since the 1980s. Much has been accomplished, but there is still much to do before thevision of better system building via reuse and domain engineering is completely achieved.

APPENDIX A

REUSE ROADMAP: A GUIDE TO THE LITERATURE OF SOFTWARE REUSE AND DOMAIN ENGINEERING

These references have been selected to give researchers and practitioners quick access to reuse and domain engineering sources, not for their historical importance.

A.1 Website

ReNews (http://frakes.cs.vt.edu/renews.html) is a Website that provides software reuse and domain engineering information including component sources, tool descrip- tions, references to books and articles, and information on workshops and conferences.

A.2 Conferences and Workshops

The main conference on software reuse and domain engineering is the International Conference on Software Reuse (ICSR). The latest, ICSR8, was held in Madrid in 2004. The next is planned for Turin in the summer of 2006. Other significant conferences on reuse have included SSR, the WISR workshops, and SAVCBS. Information on these events can be found on the ReNews website.

A.3 Architecture

M. Shaw and D. Garlan, *Software Architecture: Perspectives onan Emerging Discipline.* Prentice Hall, 1996.

L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, second ed. Addison-Wesley, 2003.

R. Kazman, M. Klein, and P. Clements "ATAM: Method for Architecture Evaluation," CMU/SEI-2000-TR-004, Soft- ware Eng. Inst., Carnegie Mellon Univ., Pittsburgh. Penn., 2000.

A.4 Domain Engineering

P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns.* Addison-Wesley, 2002.

W. Frakes, R. Prieto-Diaz, and C. Fox "DARE: Domain Analysis and Reuse Environment," *Annals of Software Eng.*, vol. 5, pp. 125-141, 1998.

K.C. Kang et al., "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Technical Report CMU/SEI- 90-TR-21, Software Eng. Inst., Carnegie Mellon Univ.,Pittsburgh, Penn., 1990.

D.M. Weiss and C.T.R. Lai *Software Product-Line Engineering: A Family-Based Software Development Process.* Addison- Wesley, 1999.

A.5 Reuse Design

J. Sametinger, *Software Engineering with Reusable Components.* New York, 1997.

C. Szyperski, D. Gruntz, and S. Murer, *Component Software: Beyond Object-Oriented Programming*, second ed. Addi-son-Wesley, 2002.

B.W. Weide, W.F. Ogden, and S.H. Zweben "Reusable

Software Components," *Advances in Computers,* vol. 33, M. Yovits, ed., pp. 1-65, 1991.

E. Gamma, R. Helm, J. Johnson, and J. Vlissides *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995.

Reuse Libraries

W.B. Frakes and P. Gandel, "Representing Reusable Software," *Information and Software Technology,* vol. 32, no. 10, pp. 47-54, 1990.

A. Mili, R. Mili, and R. Mittermeir, "A Survey of Software Reuse Libraries," *Annals Software Eng.,* vol. 5, pp. 349-414, 1998.

A.6 Generative Methods

T.J. Biggerstaff "A Perspective of Generative Reuse," *Annalsof Software Eng.,* vol. 5, pp. 169-226, 1998.

K. Czarnecki, and U.W. Eisenecker *Generative Programming: Methods, Tools, and Applications.* ACM Press/Addison- Wesley, 2000.

A.7 Programming Languages and Reuse

J. Bentley, "Little languages," *Comm. ACM,* vol. 29, no. 8, pp. 711-721.

B. Stroustrup "Language-Technical Aspects of Reuse," *Fourth Int'l Conf. Software Reuse (ICSR '96),* pp. 11-19, 1996.

I. Jacobson, M. Griss, and P. Jonsson *Software Reuse: Architecture, Process, and Organization for Business Success.* Addison-Wesley, 1997.

A.8 Reuse Management and Economics

J. Favaro, K. Favaro, and P. Favaro, "Value Based Software Reuse Investment," *Annals of Software Eng.* vol. 5, pp. 5- 52, 1998.

W. Lim, *Managing Software Reuse : A Comprehensive*

W. Frakes, R. Prieto-Diaz, and C. Fox, "DARE: Domain Analysis and Reuse Environment," *Annals of Software Eng.,* vol. 5, pp. 125-141, 1998.

O. Alonso, "Generating Text Search Applications for Databases,"

IEEE Software, pp. 98-105, 2003.

D.M. Weiss and C.T. R. Lai, *Software Product-Line Engineering: A Family-Based Software Development Process.* Addison-Wesley, 1999.

K.C. Kang, J. Lee, and P. Donohoe, "Feature-Oriented Product Line Engineering," *IEEE Software,* vol. 19, no. 4, pp. 58-65, July/ Aug. 2002.

K.C. Kang et al., "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Technical Report CMU/SEI-90-TR-21, Software Eng. Inst., Carnegie Mellon Univ., Pittsburgh, Penn., 1990.

K.C. Kang et al., "Feature-Oriented Product Line Software Engineering: Principles and Guidelines ," *Domain Oriented Systems Development: Perspectives and Practices,* K. Itoh et al., eds., pp. 29-46, 2003.

C. Atkinson et al., *Component-Based Product Line Engineering with UML.* Addison-Wesley, 2002.

H. Gomaa, *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures.* Addison-Wesley, 2004.

R. Ommering et al., "The Koala Component Model for Consumer Electronics Software," *Computer,* vol. 33, no. 3, pp. 78-85, Mar. 2000.

J. Kramer et al., "Software Architecture Description," *Software Architecture for Product Families: Principles and Guide to Strategically Reengineering the Organization for Reusable Components.* Prentice Hall, July 1998.

A.9 Reuse Measurement

W. Frakes and C. Terry, "Software Reuse: Metrics and Models," *ACM Computing Surveys,* vol. 28, no. 2, pp. 415-435, 1996.

J.S. Poulin, *Measuring Software Reuse: Principles, Practices, and Economic Models.* Addison-Wesley, 1997.

ACKNOWLEDGMENTS

The guest editors would like to thank the reviewers of the papers included in this special issue, the attendees who helped select the papers, and the reuse researchers and practitioners who responded to their survey: Sidney Bailin, Young Cho, John Favaro, Wayne Lim, Ali Mili, Bruce Weide, and David Weiss. They would also like to thank Juan Llorens and Universidad Carlos III de Madrid for hosting ICSR8.

REFERENCES

B. Stroustrup, "Language-Technical Aspects of Reuse," *Proc. Fourth Int'l Conf. Software Reuse (ICSR '96),* 1996.

J. Favaro, K. Favaro, and P. Favaro, "Value Based Software Reuse Investment," *Annals of Software Eng.,* vol. 5, pp. 5-52, 1998.

C. Krueger, "Eliminating the Adoption Barrier," *IEEE Software,*

pp. 29-31, July/Aug. 2002.

W. Frakes and C. Terry, "Software Reuse: Metrics and Models,"

ACM Computing Surveys, vol. 28, pp. 415-435, 1996.

Practice, M. Jazayeriet al., eds., pp. 31-64, 2000.

F. Buschmann et al., *Pattern-Oriented Software Architecture.*

Chichester, UK; New York: Wiley, 1996.

W. Tracz, "DSSA (Domain-Specific Software Architecture) Peda- gogical Example," *ACM SIGSOFT Software Eng. Notes,* vol. 20, no. 3, pp. 49-62, July 1995.

N. Leveson, *Safeware: System Safety and Computers.* Addison- Wesley, 1995.

B. Meyer, ".NET is Coming," *Computer,* vol. 34, no. 8, pp. 92-97, Aug. 2001.