Software Reuse in the USC System Factory Project

Walt Scacchi

Decision Systems Dept.
University of Southern California
Los Angeles, CA 90089-1421

Scacchi@pollux.usc.edu

Abstract

In this note, I briefly review and describe some of the principal techniques and mechanisms for software reuse that we have used within the USC System Factory Project. I first review our techniques for software reuse that include domain analysis and modeling, formal development methods, reverse software engineering, module interconnection formalisms, and software process reuse. I then follow with a similar review of the computational mechanisms we use to support reuse including object repositories, software generators, process-based environments, software configuration tools, and extensible systems.

Keywords: Domain analysis and modeling, formal development methods, reverse software engineering, module interconnection formalisms, process reuse, software reuse mechanisms

1 Introduction

In order to help convey an idea of our past and present efforts in software reuse in the USC System Factory Project, I have prepared this working paper. However, I do want to point out that software reuse, per se, has not been a distinct research topic for us in the SF Project. Instead, we think of software reuse as a basic strategy for improving our software development productivity. Thus, we seek to make frequent and widespread application of software reuse techniques and mechanisms in our R&D activities.

First, I will identify the categories of software reuse techniques and mechanisms that are relevant, then follow with a brief description of our efforts within each. Then, I will briefly describe some strategies for organizing software reuse activities. Overall, we have a number of research publications, available from the author, that provide more detailed descriptions of our efforts than appropriate here.

2 Software Reuse Techniques

What follows is an unordered set of techniques for software reuse that have been employed within the SF project. The notion of "technique" here implies an activity performed by one or more persons that may or may not use a systematic notation to describe or enact a software reuse strategy. Techniques appear in contrast to "mechanisms", described later, which refer to computational tools, executable processes, or other operational software artifacts that might support some reuse technique.

The reuse techniques of interest to us include domain analysis and modeling, formal development methods, reverse software engineering, module interconnection formalisms, and software process reuse.

2.1 Domain Analysis and Modeling

A few years ago, I undertook an effort to survey and categorize the various families of software systems available in the commercial and academic marketplace. I found twenty different application families covered the few thousand software offerings then available. In turn, I then had teams of graduate students conduct a systematic analysis of the operational requirements of the software systems that typified each family. I also had these students conduct a literature review of some 250 articles that covered software applications in each of the 20 families. From this analysis, I identified a small set of domain-independent software subsystems (e.g., user interface management systems, structured storage servers, etc.) that were common to two or more families. Now while this set of software subsystem components is now fairly obvious, it became clear that it was also possible that unexplored software applications (ie, potentially innovative applications) could be derived by mixing and matching components from different application families. For example, a data visualization and animation subsystem might be combined with a corporate order-entry, dispersements, and payroll software components as a skeletal framework for visualizing corporate cash flow patterns and dynamics. Of course, the significance (or lack thereof) for these new application systems is in the eye of the beholder (or customer). Nonetheless, this domain analysis of software application families did help to reveal which large-scale software components could be targeted for reuse in different application domains [Scac 89]. Also, it helped to reveal that development environments for software application families should provide easy access to rapid composibility of subsystems to support more rapid application development or prototyping.

2.2 Formal Development Methods

Actually, one purpose of the preceding domain analysis of software product families was to try to derive a set of formal specification (in the Gist specification language from USC-ISI) for the common software subsystems [Cast 86]. This effort provide to be demanding and time-consuming. Furthermore, we lacked a formal specification language support environment. Thus, the use of formal development techniques such as operational specification languages to specify common software subsystem components remains an underexplored reuse technique for us.

Nonetheless, we have investigated reuse opportunities for formal specifications. For example, in one experiment, we provided a set of development teams (5-7 graduate students in each) with

access to a hypertext-based catalog of formal specifications [Bendi 89]. The teams were given a problem statement to develop and deliver a formal specification (in Gist) and informal narrative specification within a two week period for a software system they were then to design, implement, and test. The specified systems were then implemented, and the resulting C source code varied in size from 3000 to 12000 lines of code. Thus, we considered the delivered specifications to be more than toy or textbook level problems.

We recorded the time each team spent producing their specification, the number of automatically detected errors in the delivered formal specification, and whether they used specification fragments available in the hypertext catalog [Garg 90]. Admittedly these are crude (but acceptable) measures of productivity, quality, and reuse. To our surprise, reuse of formal specification, per se, was not significantly associated with productivity or quality variations. Instead, we found that intra-team dynamics accounted for the most variance observed in productivity and quality. While these results are only suggestive, not definitive, they did begin to dissuade us from further studies of the reuse of formal specifications as a productivity or quality enhancement strategy. However, we do still believe that formal specification do contribute to improved understanding of the software applications by their developers.

2.3 Reverse Software Engineering

As part of our work in software engineering environments, we got interested in reverse software engineering (RSE) and software system re-engineering. Our focus was aimed at (a) extracting, visualizing, and restructuring architectural design representations from source code, and (b) transitioning legacy code into forms compatible with advanced software engineering environments (such as those incorporating configuration management services). A couple of companion papers describe these efforts and associated environment mechanisms in more detail [Choi 90, Choi 91].

From a reuse perspective, one reason to investigate RSE techniques is to determine to what extent extracted architectural designs reveal software reusability information. Consider the following: Assume that an extracted architectural design of a software application can be represented by a directed-acyclic graph, with nodes corresponding to application modules, and edges as module interconnections for resource exchange. Then we might, for example, seek to identify modules with high interconnectivity as candidates for reusable components. This makes more sense when many applications have their architecture extracted and logical subsystems compared. Alternatively, if we must build new generation replacement systems for older less maintainable systems, then recovered design information might serve as a guide for more rapidly analyzing and prototyping the new replacement system. However, when moving to new source code language paradigms (e.g., from imperative to object-oriented) this technique may not be as useful. Instead, we may need to develop application-specific RSE tools which produce a neutral, intermediate representation for possible reuse. We have proposed to develop such tools to some of our research sponsors to aid their proposed redevelopment of large programs in "old" programming languages into Ada.

2.4 Module Interconnection Formalisms

There is now growing interest within the software research community focussed on the rapid composition of reusable, large-grain software components/subsystems into large systems. Techniques

now called "megaprogramming" by some are representative of this interest. Basically, the idea is that there should exist a separate notation, language, or some other formalism for describing the interfaces and interconnection portals for large components that can configured into operational systems with modest effort.

Our work in this area is an outgrowth of our prior studies in developing and applying module interconnection language (MIL) concepts and mechanisms to support the evolution of configured software life cycle descriptions (ie, life cycle components, not just source code components) [Choi 89, Garg 88, Nara 87a, Nara 85, Nara 87b]. In 1985, we developed the NuMIL language for specifying how families of multi-version source code modules can be interconnected into subsystem families through well-defined resource exchange interfaces. The NuMIL notation was then adapted for use in any software description notation, formal or informal, and supported by the SOFTMAN environment since 1989 [Choi 89, Choi 90, Choi 91]. This environment in turn supports the incremental development, verification and validation, and quality assurance of software applications throughout their life cycle. We are currently restructuring SOFTMAN so as to incorporate a programmable life cycle process interface.

The NuMIL notation was also at the same time adapted to serve as a data and tool integration language for use in the DIF hypertext environment [Garg 90]. DIFConfig, as well now call it, serves as a domain-independent hypertext environment shell into which we can inegrate existing software tools or applications from other domains in order to rapidly create domain-specific hypertext environments (DSHE) [Scac 89]. These DSHE also provide hypertext and data management services to the integrated applications. We have used DIFConfig to develop DSHE for journal publication, computer-aided design (CAD/CAM), medical clinic information systems, payroll and personnel, and computer music composition applications. If one were to measure the volume of source code assembled in these DIFConfig-based environments, the measures run in the range of 50K SLOC to 250K SLOC. Further, these application environments were developed by domain specialists unfamiliar with our mechanisms, but who averaged about 120 hours of total development effort from start to sign-off.

We are currently redoing the backend to DIFConfig to utilize a newly developed distributed hypertext (DHT) service layer which will provide hypertext navigation and integration capabilities to applications, tools, or data distributed over local/wide-area network of heterogeneous information repositories [Noll 91]. The DHT service is also being added to the restructured SOFTMAN environment noted above.

2.5 Process Reuse

We have come to recognize that most of the attention directed at software reuse is directed to reuse of executable software products. However, we observe that if our interest is to improve productivity or to reduce the time to get new products out the door, we can also focus attention to improving and optimizing software production processes. However, until recently, software processes were informal, too abstract, and lacking any operational representation. Times have changed.

In order to improve and optimize software production processes, they must be observable, repeatable, measurable, enactable, and reconfigurable. In short, software processes should be reusable in order to be improvable and optimizable.

We are now developing and experimenting with formal languages and graphic notations for specifying operational software process models. These process models in turn can be used to integrate and "drive" software development environments, such as SOFTMAN noted earlier. We have developed a number of associated mechanisms for modeling, simulating, configuring, repairing, querying, and replaying software process specifications [Mi 90, Scac 91, Scac 86]. As we develop larger number of software process specifications, we will need to also develop techniques and mechanisms for indexing process specifications as well. Last, we should also note that process specification techniques and mechanisms may be applied to domains other than software production, such as manufacturing, technology transition, training, and others [Scac 89].

3 Software Reuse Mechanisms

The following is an unordered set of computational mechanisms for software reuse that have been employed or mentioned by various researchers:

3.1 Object Repositories and Catalog Servers

Most of the software reuse techniques described above implicitly expect the availability of some sort of software component or artifact repository. We have experimented with various hypertext mechanisms as a way to catalog, browse, query, and access various reusable software entities. At present, our attention is directed to construction of a distributed hypertext service layer (noted above) that will enable autonomous, distributed heterogeneous repostories to be integrated and accessible through a common hypertext-based communication protocol [Noll 91]. Thus, this service layer might enable various repositories of software entities on a wide-area network to be accessed, browsed, and so forth as if they were part of a global or corporate-wide repository.

On the other hand, there are still some substantial problems that current repositories and catalog servers do not address very well. These problems include (a) naming reusable entities with semantically meaningful names or part numbers; (b) discovering whether there exists a reusable entity that satisfies some request, specification, or semantic signature; and (c) determining which repository to search for certain types of reusable entities. At this time, we are investigating part naming techniques developed for use in group technology as a way to address (a). For (b) and (c), we think that "intelligent gateways" or "distributed search agents" may be needed as extensions to the DHT service layer. But these are still preliminary hypotheses.

3.2 Software Generators

Software generators, as the name suggests, are programs that produce other programs. In this regard, software generators are a kind of "meta-reusable" software component, since the programs they produce might themselves be treated as reusable components. The most common example of software generators are parser and lexical analyzer generators. Of course, other kinds of generators have been produced including code generator generators, full compiler generators, report generators, and various application generators. We have developed language-directed editor generators,

user interface generators, formal specification (Gist) generators, and spreadsheet application generators [Cast 86]. As these generators all produce operational stand-alone programs in the range of a 1K-25K+ lines of code, such programs become more interesting when they can be generated as complete subsystems that can be rapidly composed or integrated into larger systems or environments. We have some experience with this in the SOFTMAN environment where we can generate language-directed editors for new languages that are easy to integrate into an existing or new SOFTMAN environment instance. We have also investigated other meta-tools and generators of software development environments as well [Karr 91].

However, we are also interested in exploring the idea of cascading different software generators together, so that the output of one generator becomes the input to one or more other generators. For example, when we built a spreadsheet application generator a few years ago, its input specification language was a subset of the Gist language. The generator in turn transformed the input specification into a working program approximately 10 times the size of the input specification (as measured by number of statements–admittedly a crude measure). We also developed a Gist generator that accepted a structured, form-based informal language as input, that in turn paraphrased the input into a different Gist subset. In this case, the output-input ratio was roughly 3-1. Finally, in a systematic study of the specification and implementation of a dozen software systems using Gist and C, we observed a C-Gist ratio of between 25-1 and 30-1. Thus, the hypothesis I then derived was that if we could restructure the Gist specification subsets used by the two generators, it might therefore be possible to connect (or cascade) the specification generator to the spreadsheet application generator, then expansion ratios of 30-1 or more might be possible. Further, if the Gist subsets could be expanded to the full Gist language, and another intermediate generator added to handle the additional constructs, then it might be possible to demonstrate a software generator cascade that could produce programs with an expansion ratio of 100-1 or more. However, the students who were working with me on this project graduated and took industrial positions before the project was completed. Thus, I think of domain-specific cascaded software generators as still a promising mechanism for research.

3.3 Process-Based Environments and Interfaces

We think that the reuse of software processes is emerging as an important technique for software production. Accordingly, we have development a number of mechanisms for exploring the value of this technique. Specifically, we have developed a knowledge-based environment for modeling and simulating complex software engineering processes [Mi 90]. The processes of greatest interest to us are those that involve multiple development agents acting in multiple, sometimes overlapping roles, who are assigned to perform a web of interrelated tasks with limited resources. Our environment, called the Articulator, has been used to model and analyze software development processes in practice by some of our industrial sponsors, as well as those which we practice in the SF Project. We have fund these software process models can be used for planning, training or guiding, monitoring, and improving software production processes. Further, through knowledge-based simulation and query interfaces, we can symbolically execute modeled processes, both forward and backward. Thus, we can configure a process model, simulate its development progress, stop and reply it, back it up to some prior state, reconfigure the state then continue on a new path of progress.

In addition, we have also developed what we call a process-based user interface (PBI) for software

development environments [Peiw 91]. With PBI, we can use a process model constructed with the Articulator to configure the development environment to directly support the process. That is, PBI adds a capability to add "process integration" to development environments or environment frameworks (e.g., HP Softbench) that provide data and/or control integration mechanisms. We are currently developing a PBI for the SOFTMAN environment which also incorporates or supports the other software reuse mechanisms described in this memo.

3.4 Software Configuration Tools

As noted in the discussion of module interconnection formalisms, we have developed a small number of software configuration mechanisms. DIFConfig is a primary example of such a mechanism that supports the development of hypertext-based application environments that can integrate large-grain software components. Our experience with DIFConfig has been very promising to date. However, the current DIFConfig implementation, based on the original DIF hypertext backplane, lacks capabilities we now consider desirable. Thus, we are currently directing effort to extending DIFConfig to support the DHT service layer, as well as to incorporate other software reuse mechanisms described in this section.

3.5 Extensible Software Systems

Objects within a class hierarchy, subclass specialization, polymorphism and inheritance are all elements of extensibility available in most object-oriented program development systems. Software development tools that support extensibility directly may therefore be appropriate as part of a reusable software environment. We have developed an extensible tree/graph editor (TGE) as an example of such a tool [Karr 90]. With TGE, it is possible to construct domain-specific tree or graph editors. The resulting editors are developed by specializing the classes of objects and functions provided in the base tree or graph editors. This development technique has the advantage of that the the target editor is incrementally built from an already operational editor. This in turns means that the emerging target editor is always operational throughout the development process. This ability to execute and try out an emerging editor during development of course provides an excellent source of feedback during early prototyping stages. With our TGE, we have developed about a dozen tree/graph editor applications which we have been able to readily integrate into a variety of application environments, including SOFTMAN. Thus, extensible systems such as TGE provide capabilities similar in ways to those of software generators.

4 Summary

In this note, I have briefly described some of the major techniques and mechanisms for software reuse that we use as a regular part of our research and development activities. Although software reuse has not been an explicit research focus for us in the System Factory project, we believe that it will increasingly become part of normal development practices, at least within the research community.

5 Acknowledgements

Work describe in this report is part of The USC System Factory project. This work was supported through contracts and grants with ATT Bell Laboratories, Northrop Corporation, Office of Naval Technology through the Naval Ocean Systems Center, and Pacific Bell.

References

[Bendi 89] Bendifallah, S. and W. Scacchi. "Work Structures and Shifts: An Empirical Analysis of Software Specification Teamwork." Proc. 11th. Intern. Conf. Software Engineering, ACM, 1989, pp. 260-270.

[Cast 86] Castillo, A., S. Corcoran, and W. Scacchi."A Unix-based Gist Specification Processor: The System Factory Experience." Proc. 2nd. Intern. Conf. Data Engineering, 1986, pp. 582-589.

[Choi 89] Choi, S., and W. Scacchi. "Assuring the Correctness of Configured Software Descriptions." Proc. 2nd. Intern. Workshop Software Configuration Management ACM Software Engineering Notes, 17(7) (1989), 67-76.

[Choi 90] Choi, S.C. and W. Scacchi. "Extracting and Restructuring the Design of Large Systems". IEEE Software 7, 1 (1990), 66-71.

[Choi 91] Song Choi and Walt Scacchi. "SOFTMAN: AN Environment for Forward and Reverse Computer-Aided Software Engineering". Information and Software Technology, to appear (1991).

[Garg 88] Garg, P.K. and W. Scacchi."A Software Hypertext Environment for Configured Software Descriptions." Proc. Intern. Workshop on Software Version and Configuration Control, January, 1988, pp. 326-343.

[Garg 90] Garg, P.K. and W. Scacchi. "A Hypertext System to Manage Software Life Cycle Documents". IEEE Software 7, 3 (1990), 90-99.

[Karr 90] Karrer, A. and Scacchi, W. "An Extensible Object-Oriented Tree/Graph Editor". ACM SIGGRAPH Symp. on User Interface and Software Technology, October, 1990, pp. 84-91.

[Karr 91] Karrer, A. and Scacchi, W. "Meta-Environments for Software Development". submitted for publication, October (1991).

[Mi 90] Mi, P. and W. Scacchi. "A Knowledge Base Environment for Modeling and Simulating Software Engineering Processes". IEEE Trans. Knowledge and Data Engineering 2, 3 (1990), 283-294.

[Peiw 91] Peiwei Mi and Walt Scacchi. "Process Integration for CASE Environments." submitted for publication, May (1991).

[Nara 87a] Narayanaswamy, K. and W. Scacchi. "A Database Foundation to Support Software System Evolution". J. Systems and Software 7 (1987), 37-49.

[Nara 85] Narayanaswamy, K. and W. Scacchi. "An Environment for the Development and Maintenance of Large Software Systems." Proc. SOFTFAIR II, 1985, pp. 11-25.

[Nara 87b] Narayanaswamy, K. and W. Scacchi. "Maintaining Configurations of Evolving Software Systems". IEEE Trans. Soft. Engr. 13, 3 (1987), 324-334.

[Noll 91] Noll, J. and W. Scacchi. "Integrating Diverse Information Repositories: A Distributed Hypertext Approach". Computer 24, 12 (1991), to appear.

[Scac 89] Scacchi, W. "On the Power of Domain-Specific Hypertext Environments". J. Amer. Soc. Info. Science 40, 3 (May 1989), 183-191.

[Scac 91] Scacchi, W. "Understanding Software Productivity: Towards a Knowledge-Based Approach". Intern. J. Soft. Engr. and Know. Engr. 1, 3 (September 1991).

[Scac 86] Scacchi, W., S. Bendifallah, A. Bloch, S. Choi, P. Garg, A. Jazzar, A. Safavi, J. Skeer, and M. Turner. Modeling System Development Work: A Knowledge-Based Approach. Computer Science Dept., USC, 1986. working paper SF-86-05.

6 Biography

Walt Scacchi received a B.A. in Mathematics, a B.S. in Computer Science in 1974 at California State University, Fullerton, and a Ph.D. in Information and Computer Science at University of California, Irvine in 1981. He is currently an associate research professor in the Decision Systems Dept. at USC. Since joining the faculty at USC in 1981, he created and continues to direct the USC System Factory Project. This was the first software factory research project in a U.S. university. Dr. Scacchi's research interests include very large scale software production, knowledge-based systems for modeling and simulating organizational processes and operations, CASE technologies for developing large heterogeneous information systems, and organizational analysis of system development projects. Dr. Scacchi is a member of ACM, IEEE, AAAI, Computing Professionals for Social Responsibility (CPSR), and Society for the History of Technology (SHOT).