Experience with Software Architectures and Configured Software Descriptions

Walt Scacchi	James S. Choi
Institute for Software Research	Computer Science Dept
University of California, Irvine	California State University, Fullerton
Wscacchi@ics.uci.edu	Sjchoi@cse.fullerton.edu

Submitted to ***WESAS***, Irvine, CA, May 2000.

Introduction

In this position paper, we highlight some of the things we have learned over the past 15 years in our experience with software architectures. Much of what we have learned results from our experience in the specification, design, implementation and evolution of software engineering environments and process-driven software environments. Along the way, we have developed or used a variety of alternative architectural notations to support these efforts. We also have employed architectural design concepts and notations to specify, "code" and evolve a variety of configured software descriptions, including software life cycle documents, software hypertexts, software processes, and others. In this regard, we have found it useful to explore alternative schemes for combining software architecture concepts, techniques, notations and tools with those for software configuration management. Accordingly, we will highlight some of our experiences in these areas or topics.

Architecture Experience with Software Environments

The architectural design of software environments appears in many forms throughout the software engineering community. Three recurring forms are (a) conceptual, (b) logical, and (c) concrete or executable. However, these three forms are not synonymous and are employed to specify different kinds of architectural information.

Conceptual architectures of software environments are the most pervasive and widely used. They are graphically or visually rendered as undirected/directed attributed graphs (e.g., box and arrow diagrams) whose syntax and semantics are vague, ambiguous or unstated. They generally always fit on one page/slide. As a result, they are most effective as "communication devices" for group discussion, presentation or publication. Nonetheless, they also may obscure technical details that can lead to things like architectural mismatches. Thus, one cannot trust conceptual architectural designs as anything more than communication media or as neat pictures that *suggest* technical directions for system development.

Logical architectural designs of software environments are usually specified with a language-based architectural design notation, such as a module interconnection language (MIL) in early efforts, or architectural design language (ADL) in more contemporary efforts. Syntactic constructs and semantic representations are employed to specify components, connections/connectors, and their compositions, as well as other properties such as complex behavioral constraints as pre/post conditions on components [e.g. Narayanaswamy and Scacchi 1987a,b]. All of the software environments we have seen or have developed that were specified with MILs/ADLs are of moderate complexity and scale (e.g. tens of components, though some components may be of substantial granularity, like a commercial DBMS). However, experience with the explicit specification of the architctural design of large software systems involving hundreds, thousands, or tens of thousands of components (or modules) is rare. Furthermore, the use of MILs/ADLs to specify behavioral constraints beyond component invocation, signalling (e.g., post error notification), or termination conditions on such large systems appears to be impractical, since "mere mortals" (i.e., average programmers, not those with Ph.D's in Computer Science) are generally assigned to developed large systems. Thus, MILs/ADLs are still an unproven technology for very large systems which nonetheless exist and continue to be developed. Instead, we believe that very large systems require the use of thin or minimal MILs/ADLs to specify a small set of high value relations, such as simply structural relations that can be employed constructively at design time, or automatically extracted at compile/run-time [Choi and Scacchi 1990, 1991].

Concrete architectural designs deal with configurations of executable software. Executable software can be broadly applied to include interpreted, compile-time or run-time versions of source code. Concrete software designs are often involve multiple languages and associated information that persists with the software. For example, network file systems and people within designated administrative authority (or "user-privilege" sub-domains) frequently use version control systems/notations to specify which version of what components, within one or more designated directories/ repositories, are to be compiled/updated/built using some resource configuration file(s), subject to some condition(s). As a result, multiple concrete architectures can follow from a single logical architectural design. Unfortunately, some of these configurations may give rise to anomalous or untractable conditions during their execution (e.g., the "blue screen of death" on MS Windows platforms). Conversely, multiple logical designs may be derived ("reverse engineered") from a single concrete architectural configuration [Choi and Scacchi 1990, 1991]. Thus, the mapping or transformation from logical to concrete architectural designs are not necessarily monotonic, and thus must be viewed as possibly entailing non-monotonic transformations [cf. Choi and Scacchi 1998]. Such a condition also points to potential problems when using MILs/ADLs that require the specification of complex behavioral constraints, unless formal correctness properties can be articulated and proved to support non-monotonic transformations.

Experience with Configured Software Descriptions

We have also found it useful to apply software architectural concepts, techniques, notations and tools to manage the systematic design, implementation and evolution of many other kinds of software artifacts. These include software life cycle documents [Choi and Scacchi 1991, 1998], multi-version software source code hypertexts [Garg and Scacchi 1988, Noll and Scacchi 1997], software development processes [Mi and Scacchi 1992, 1996, Scacchi 1999, Choi and Scacchi 2000], wide-area software repositories [Noll and Scacchi 1991], software acquisition processes [Scacchi and Boehm 1998], and the specification of networked/virtual enterprises [Noll and Scacchi 1999, Scacchi and Noll 1997]. The composing and configuring the architectures of these kinds of software artifacts can be (and has been) supported by MILs/ADLs that emphasize specification of structural relations as essential, while minimizing the specification of behavioral constrations. This represents an engineering trade-off for what is most practical versus what is most formally robust.

What emerges from these experiences is that architectural design and configuration management technologies can be combined, integrated and interoperated for mutual benefit, as well as applied to any kind of configured software description. This means that informal (e.g., structured narrative documents), semi-structured (e.g., software hypertexts), and formal (e.g., semantic network data/system models [Mi and Scacchi 1996]) can be designed, composed, configured and evolved as multi-version, multi-configuration software artifacts. It also enables representations of these configured software descriptions to be navigationally traversed using hypertext/Web-based technologies. Further, it also allows for the interpretation or incremental/JIT compilation of executable software artifacts (programs, tool scripts, process enactment instances, etc.) as a consequence of navigational traversal (via "software web surfing") [Noll and Scacchi 1997, 1999]. This in turns enables a form of reconfigurable execution dynamism (e.g., process dynamism [Noll and Scacchi 1999]) which therefore supports basic forms of dynamic compile-time or run-time adaptability.

Conclusions

In this position paper, we have highlighted some of the experiences we have encountered through the development, use and evolution of various software environments. We welcome the opportunity to present and discuss these experiences and others at the WESAS workshop.

References

J.S.C. Choi and W. Scacchi. Extracting and Restructuring the Design of Large Software Systems. *IEEE Software*, 7(1): 66-73, January 1990.

J.S.C. Choi and W. Scacchi. SOFTMAN: An Environment for Forward and Reverse Computer-Aided Software Engineering, *Information and Software Technology*, 33(9): 664-674, November 1991.

J.S.C. Choi and W. Scacchi. Formalization and Tools Supporting the Structural Correctness of Software Life Cycle Descriptions, *Proc. IASTED Conf. on Software Engineering*, Las Vegas, NV, 27-34, October 1998.

J.S.C. Choi and W. Scacchi. Modeling, Analyzing and Simulating Software Acquisition Process Architectures. (submitted for publication), February 2000.

P.K. Garg and W. Scacchi. A Hypertext Environment for Managing Configured Software Descriptions, *Software Version and Configuration Control*, pp. 326-343, B.G. Teubner, Stuttgart, FRG, (January 1988).

P. Mi and W. Scacchi. Process Integration in CASE Environments, *IEEE Software*, 9(2): 45-53, (March 1992). Reprinted in *Computer-Aided Software Engineering (CASE)*, Second Edition, Eliot Chikofsky (ed.), IEEE Computer Society, (1993).

P. Mi and W. Scacchi. A Meta-Model for Formulating Knowledge-Based Models of Software Development, *Decision Support Systems*, 17(4):313-330, (1996).

K. Narayanaswamy and W. Scacchi. Maintaining the Configuration of Evolving Software Systems, *IEEE Trans. Software Engineering*, 13(3): 324-334, March, 1987a.

K. Narayanaswamy and W. Scacchi. A Database Foundation for Supporting the Evolution of Large Software Systems, *J. Systems and Software*, 7(1): 37-49, 1987b.

J. Noll and W. Scacchi. Integrating Heterogeneous Information Repositories: A Distributed Hypertext Approach, *Computer*, 24(12): 38-45, (December 1991).

J. Noll and W. Scacchi. Supporting Distributed Configuration Management in Virtual Enterprises. appears in R. Conradi (ed.), *Software Configuration Management*, Lecture Notes in Computer Science, Vol. 1235, Springer-Verlag, New York, 142-160, 1997

J. Noll and W. Scacchi. Supporting Software Development in Virtual Enterprises, *Jour. Digital Information*, 1(4), February 1999.

W. Scacchi. Experience with Software Process Simulation and Modeling. J. Systems and Software, 46(2/3):183-192,1999.

W. Scacchi and B.E. Boehm. Virtual System Acquisition: Approach and Transitions, *Acquisition Review Quarterly*, 5(2):185-216, Spring 1998.

W. Scacchi and P. Mi. Process Life Cycle Engineering: Approach and Support Environment, *Intelligent Systems in. Accounting, Finance, and Management*, 6(1):83-107, (1997).

W. Scacchi and J. Noll. Process-Driven Intranets: Life Cycle Support for Process Reengineering, *IEEE Internet Computing*, 1(5):42-49, (1997).