Software Component Technologies and Space Applications

Don Batory
Department of Computer Sciences
The University of Texas
Austin, Texas 78712

Abstract

In the near future, software systems will be more reconfigurable than hardware. This will be possible through the advent of software component technologies, which have been prototyped in universities and research labs. In this paper, we outline the foundations for these technologies and suggest how they might impact software for space applications.

1 Introduction

Software component technologies will fundamentally change the way complex and customized software systems will be designed, developed, and maintained. Well-understood domains of software (e.g., avionics software, communication networks, operating systems, etc.) will be standardized as libraries of plug-compatible and interoperable components. A software system in these domains (e.g., a particular avionics system, a particular operating system, etc.) will be specified as a composition of components. High-performance source code that implements these systems will be generated automatically. Application developers will purchase component libraries for the domains of interest, and will configure components to build the target systems/platforms that their applications need. The evolution of software, once a formidable problem, is radically simplified: an updated version of a system is defined as a composition of components and its software is generated automatically. It is in this manner that future software engineers will leverage off of existing componentry to "mass produce" complex and customized software quickly and cheaply.

For this vision to become a reality requires basic changes in the way we understand and write software. First and foremost, a software component technology requires us to address the following:

- **encapsulation** - what should a building block (i.e., component) of software systems encapsulate?
- **composition** - what does composition mean?
- **paradigm** - what model of programming supports software component technologies?
- **scalability** - how can large families of systems be expressed by a small number of components?
- **verification** - how can one verify that a composition of components is consistent and implements the specifications of the target system?

Answers to these questions lie at the confluence of a number of independent research areas: transformation systems, object-oriented programming, parameterizing programming, domain-specific compiler optimizations, and domain modeling and the design of reusable software. Research on software system generators lies at the heart of this intersection, where specific and practical answers to these questions have been found.

2 A Paradigm Shift

The evolution of customized software is the bane of most projects: it is difficult to achieve and is horrendously costly. There is always the need to develop new variants of existing systems, each variant/version offers new features that are specific to the class of applications that are to be supported. But often the effort needed to make even minor changes to a system is far out of proportion to the changes themselves [Par79].

The problem is that source code is the most detailed and concrete realization of a software design. The most critical changes (and hence the most important evolutionary changes) to a software system are modifications to its design. Minimal design changes often require major software rewrites. Rather than maintaining and evolving source code, an alternative is to maintain and evolve the *design* and to *generate* the corresponding source code automatically. This is the concept of *design maintenance* [Bax92].

Design maintenance asserts that the designs of software systems are quantized; there are primitive components of software design in every domain. A component encapsulates a domain-specific capability that software systems of that domain can exhibit. The design of a software system is therefore expressed as a composition of components, where a composition defines the set of capabilities that a target system is to have. Furthermore, the evolution of a system's design occurs in quantum steps and these steps correspond to the addition or removal of domain-specific capabilities (i.e., components) from the target system.

Design maintenance has two important implications. First, conventional methods of software design must change because they view software systems as one-of-a-kind products. Reusing previous designs or source code is largely an ad hoc and fortuitous activity. Design maintenance, in contrast, requires the identification of primitive components of design for a large family (or domain) of software. A primitive component, by definition, is reusable because it is used in the design of many family members. *Domain modeling* is the name given to software design methodologies that identify primitive components of software designs for a specific domain [Pri91, Gom94, Bat95a].

A second implication is that a primitive component of software design need not correspond to a primitive code module or package in a generated system. In general, the introduction of a component to a system's design might require incremental modifications to many parts (e.g., object-oriented classes) of a system's software. Furthermore, the modifications that a component makes to the source code of one system might be different than that made to another; such differences arise because certain domain-specific optimizations could be applied to one system, but not in the other. Thus a component must encapsulate more than just algorithms: it must also encapsulate *reflective* computations, i.e., domain-specific decisions about when to use a particular algorithm and/or when to apply a domain-specific optimization. For most domains, reflective computations are critical for generating efficient code [Bat93].

What programming paradigm supports such componentry? The rallying cry of object-orientation is that "everything is an object". Object-oriented design methodologies and programming languages are indeed powerful, but they are insufficient for software components. A programming paradigm that has been found to encompass object-orientation, in addition to providing the generality needed, is that of *program transformation systems*. The rallying cry of transformation systems is "everything is a transformation", or more specifically a *forward refinement program transformation (FRPT)*. The connection between components and FRPTs is direct: an FRPT elaborates a high-level program by introducing details (e.g., source code) that efficiently implement a domain-specific capability. Such elaborations can occur in many parts (e.g., classes) of a system's software. Moreover, an inherent part of an FRPT is the ability to perform reflective computations, so that only the most efficient algorithms are generated. Composing components is equivalent to composing transformations [Bax92, Bat92].

The key ingredient that enables components to be composed is due to a disciplined design that standardizes the abstractions (and their programming language interfaces) of a domain. Simply put, domain modeling ensures that components are designed to be interoperable, interchangeable, and plug-compatible and thus can be used as software building blocks; components with ad hoc interfaces that are not interoperable, interchangeable, and plug-compatible are not building blocks.

Among the benefits of software componentry is that few components are actually needed to assemble large families of systems. We expect most domain-specific libraries to have a few hundred components, where domain experts can easily identify components to be used (without requiring elaborate library classification and searching methods). Another benefit of software componentry is that there are simple algorithms to determine automatically if a composition of components is consistent and that it implements the specifications of a target system. While demonstrating consistency falls short of formal verification, it is an major step forward in making software system generation practical [Per89, Bat95b].

Software component technologies and generator technologies have been developed for the domains of avionics, database systems, file systems, network protocols, and data structures. Related composition/encapsulation technologies in software architectures are [Gor91, Per92, Gar93]. Readers who are interested in the technical details of these discussions are urged to consult the cited references.

3 Relevance to Software Development for Space Applications

In the following, I address the community of software developers for space applications. However, I admit that there is very little in my comments that are specific to space applications; the problems that I address and the benefits that can be reaped are applicable to software in general.

The main obstacles I foresee in the promulgation of software component technologies and software system generators are not technical in nature. To be sure, there are plenty of difficult technical problems ahead, but I am confident that these problems are solvable. My intuition for this not-very-bold statement is that domain modeling takes a retrospective view of software systems that have been built. Thus, solutions to thorny design problems have already been devised in a multiplicity of contexts. The activities of domain modeling - the basis of software component technologies - are to show how these specific solutions fit into a more general (i.e., building blocks) context. It is not the case that entirely new solutions to domain-specific problems (e.g., space applications) must be invented for software component technologies to work. Very little "invention" of new algorithms, etc. is needed. Hence my optimism.

The real challenge will be the acceptance of software component technologies by the space application community. The primary obstacle is that programmers and system designers are reluctant to change the way they understand and view problems in software. More specifically, this is the "not-invented-here" syndrome. If software componentry for space applications were invented in-house, it would have a much greater chance of being used. But even in-house development would be a major step from conventional approaches.

The reluctance to change has its consequences: researchers will be encouraged to seek a "silver bullet" that will miraculously solve intractable problems that have been brought on by traditional and established methods of software production. The difficulties of software evolution; the infeasibility of implementing competing, possibly radically different, designs for evaluation; the inexpensive development of product families are examples. Experience has shown that enough (minimal) progress and enough clever ideas will be demonstrated by researchers to keep the "silver bullet" hopes of software managers alive for years to come. However, I am skeptical that incremental progress will ever lead to a satisfac-

tory and economical solution. To address the major problems of software development today will ultimately require a paradigm shift.

Paradigm shifts occur when there is a general perception that major benefits will ensue. The shift from structured programming and C-like programming languages to object-oriented design methods and programming languages is being paved by a wide spectrum of realized benefits and good salesmanship of object-orientation. The same will be needed for software component technologies. The benefits of componentry are real and substantial, but are not yet that well understood or appreciated. Not surprisingly, number of advocates for software componentry needs to be enlarged.

Despite my enthusiasm for software componentry, I don't believe software component technologies are silver bullets. These technologies do not solve problems, but only simplify some problems (e.g., evolution). For example, there are there are many performance-related parameters in avionics software whose values must be determined through extensive testing and simulation. Avionics source code with such performance-related parameters is typically easy to generate. However, how one determines the values to be assigned to these parameters (e.g., aircraft-specific parameters) does not seem to be fully automatable, and the tried-and-true processes of testing and simulation still need to be performed. Thus, many of the existing activities of software development will still remain.

Another point to be made is that most "new" systems always include new and unprecedented functionality. It has been estimated that upwards of 80% of a "new" system can be built from available components. This means that 20% of the "new" system will need to be added. While a factor of five reduction in the amount of software to be written is a substantial savings, software development will certainly not cease.

But there will also be unique opportunities that software component technologies provide that would otherwise be difficult or impractical. For example, synthesis from specifications makes it feasible to evaluate radically different software designs. As another example, self-tuning and self-reorganizing software is possible: components can be added to systems to monitor their performance. Periodically, the system can reconfigure itself automatically, based on known usage patterns, to enhance its performance.

In conclusion, if software evolution, the cost-effective creation of product families, the need to experiment and retrofit system designs, and improving programmer productivity are critical to future software for space applications, then the design and use software component technologies should be made a top priority.

4 References

[Bax92] I. Baxter, "Design Maintenance Systems", *CACM* April 1992, 73-89.

[Bat92] D. Batory and S. O'Malley, "The Design and Implementation of Hierarchical Software Systems with Reusable Components", *ACM TOSEM*, October 1992.

[Bat93] D. Batory, V. Singhal, M. Sirkin, and J. Thomas, "Scalable Software Libraries", *ACM SIGSOFT* 1993.

[Bat95a] D. Batory, L. Coglianese, M. Goodwin, and S. Shafer, "Creating Reference Architectures: An Example From Avionics", *ACM SIGSOFT Symposium on Software Reusability*, Seattle, 1995, 27-37.

[Bat95b] D. Batory and B.J. Geraci, "Validating Component Compositions in Software System Generators", Department of Computer Sciences, University of Texas at Austin, TR-93-03, February 1995.

[Gar93] D. Garlan and M. Shaw, "An Introduction to Software Architecture", in *Advances in Software Engineering and Knowledge Engineering*, Volume I, World Scientific Publishing Company, 1993.

[Gom94] H. Gomaa, L. Kerschberg, V. Sugumaran, C. Bosch, and I. Tavakoli, "A Prototype Domain Modeling Environment for Reusable Software Architectures", *Third International Conference on Software Reuse, Rio de Janeiro*, November 1-4, 1994, 74-83.

[Gor91] M.M. Gorlick and R.R. Razouk, "Using Weaves for Software Construction and Analysis", *Proc. ICSE* 1991, 23-34.

[Par79] D.L. Parnas, "Designing Software for Ease of Extension and Contraction", *IEEE Transactions on Software Engineering*, March 1979.

[Per89] D.E. Perry, "The Logic of Propagation in The Inscape Environment", *ACM SIGSOFT 1989*, 114-121.

[Per92] D.E. Perry and A.L. Wolf, "Foundations for the Study of Software Architecture", *ACM SIGSOFT Software Engineering Notes*, October 1992, 40-52.

[Pri91] R. Prieto-Diaz and G. Arango (ed.), *Domain Analysis and Software Systems Modeling*, IEEE Computer Society Press 1991.