Toward an Execution Model for Component Software

Michael Franz
Department of Information and Computer Science, University of California,
Irvine, CA 92697-3425

Abstract

The notion of component-oriented programming is largely based on a model of software distribution: if a certain software artifact can be distributed independently of other similar artifacts, and yet be smoothly integrated into an environment composed of such artifacts, then we call it a component. Unlike established areas of computer science, such as object-oriented programming, component-oriented programming has no coherent programming model associated with it at this time, nor is there a general agreement as to what the run-time structure of component-based systems should be.

Basing a whole new technology on the "middle" ground of software distribution is at once strange and risky. A host of potential misunderstandings and incompatibilities will arise unless the existing model is soon extended twofold: On one hand, the mechanics of component interaction need to be brought semantically closer to programming languages. On the other hand, components need to be mapped onto appropriate run-time architectures.

Our research is focused on the latter part of the problem: we are developing techniques that enable systems composed of independently-distributed platform-independent software components to function as efficiently as monolithic application programs. A key element of our strategy is profile-guided run-time code generation, provided as a central system service, which allows to genuinely integrate dynamically loaded extensions with the existing code base.

1 Introduction

When discussing different programming paradigms, we usually base our taxonomy on the underlying design methodology, or, less subtly, on the programming language being used in a design. Strangely enough, this is no longer true when it comes to component-oriented programming. In this area alone, the *software distribution model* becomes the primary classification criterion. On one hand, this is hardly surprising, given that, for the most part, software distribution has in the past been based almost exclusively on the model of the *monolithic application* *program*, from which components represent a striking departure. On the other hand, however, this development is alarming, as it might lead to a temporary obfuscation of the issues at hand: we might *appear* to be talking about the same concerns, while we are basing our reasoning on conflicting programming models. Unfortunately, however, it is the programming model that is the foundation of a software architecture, and not its distribution model.

Let us then examine different software architectures representing different paradigms, in terms of their programming model, distribution model, and execution model (Figure 1): The shift from conventional to *object-oriented* programming has revolutionized only the programming model, but left distribution and execution unchanged. For the most part, object-oriented programming has not eliminated application programs; only a negligible fraction of object-oriented systems are based on the distribution of individual objects that execute in an object-centered host environment.

In contrast, *modular* systems have distribution and execution models that depart from the traditional blueprint. Discarding the notion of an application program, they consist of a hierarchy of modular building blocks (usually a directed acyclic graph), each of which can conceptually serve as a library to one set of modules, while simultaneously acting as a client of another set of modules. Hence, designers of application-level software can “factor out” common functionality so that it can be shared even by independent application packages. The operating system itself is usually also represented as a collection of modules, adding to the model’s uniformity. In recent years, some elements of this modular execution model have found their way into mainstream operating systems by way of the dynamic-link-library concept.

Component-oriented systems, at last, are so far identified only by their distribution model. There are no generally accepted programming models at this time, and several important issues are still unresolved. For example, it is an open question how the library base underneath a component-oriented system can be evolved without invalidating already existing components: An alternative to changing the library directly might be to add new features in the guise of extra components that would then be shared by other components. However, this in turn requires a mechanism by which different components can share common behavior – and such a mechanisms should also be intuitive for the programmer.

Just as there is no well-established programming model for component-oriented software, there is also no standard execution model. Some of the initial solutions that are currently appearing in the marketplace are extensions of the dynamic-link-library concept. This gives components a role similar to those of modules, but with a coarser granularity of interaction and a looser coupling than normally found in modular systems. Most of the commercial offerings so far have no proper programming-language support for modularization, but require error-prone manual coding of component interaction protocols.

conventional program

programming model:	monolithic program (possibly divided into several source files)
distribution model:	application program (if multiple compilation units exist, then they are combined statically by a linker)
execution model:	monolithic program requesting the services of the operating system through supervisor calls

framework-based object-oriented program

programming model:	extension of an application framework (customization of a semi-finished product)
distribution model:	application program (framework-parts and user-extensions are statically linked to form an application)
execution model:	monolithic program requesting the services of the operating system through supervisor calls

modular system

programming model:	extension of a module hierarchy (shared code between applications)
distribution model:	individual modules that import services from and export services to other modules; in practice, a standard module library is assumed to be present on every machine
execution model:	modules calling each other *directly*, *the operating system itself* is represented as a collection of modules, modules form a global hierarchy with shared sub-graphs,

component-oriented system

programming model:	?
distribution model:	independent components that interact with other components and with a shared library base
execution model:	?

Figure 1: Programming, Distribution and Execution Models for Various Software-Architecture Paradigms

One of the reasons why the modular-programming concept hasn’t simply been extended to component-oriented systems, but a coarser granularity been chosen instead, is the relative inefficiency of modular code. Increasingly super-scalar processors require sophisticated optimization techniques that perform the better the larger the “field of view” of the code generator is. In modular systems, this “field of view” stops at the module boundary, so that, for example, a common subexpression appearing in two different modules cannot be recognized as such and not be eliminated.

Our research attempts to overcome the performance deficiencies caused by modularization, allowing to extend the established model of modular software to component-oriented programming without incurring the associated penalties. Instead of creating a completely new programming model for component software, we equate components with modules, and strive to make their use more efficient. Hence, the goal of our work is to improve on the execution model of modular software. We do not attempt to devise any new programming models, but leave this task to others. However, by basing our work on a modular system, namely the *Oberon System* [1], we give it a firm foundation with adequate language support [2] that may well turn out to be wholly sufficient for most component-oriented software architectures.

2 An Architecture Centered Around Run-Time Code-Generation

In an ideal component-oriented system, software distribution should have the following properties:

- It should be *instantaneous*, i.e. happen over a network. Consequently, the distribution format needs to be highly ***compact***.
- The distributed software should be in a format that is *independent of the eventual target machine*. Hence, some kind of ***intermediate representation*** is required.
- New components should be able to enhance the capabilities of the executing system *immediately*, and not only after restarting the system. This means that some form of ***dynamic linking*** needs to be supported.
- Software should run efficiently. This rules out interpreted execution of the intermediate representation and mandates ***on-the-fly compilation***.
- No performance penalty should be associated with modularization. Elimination of modularization overhead requires ***aggressive optimization across component boundaries***.

We are developing a run-time architecture that provides all of these capabilities simultaneously. At the heart of our system is a machine-independent intermediate representation that is used for encoding independently-compiled components. This representation, “Slim Binaries” [3, 4], is significantly more compact than either compressed source code or compressed object code.

The second key element of our architecture is *run-time code generation*, provided as a central system service. When a component is first activated, the system generates native code on-the-fly from the Slim Binary representation. This occurs at remarkable speeds: since Slim Binaries are more compact than traditional binaries, less time is spent on input/output operations and can instead be diverted to code generation.

However, the duties of the code generator do not end with this initial code generation. While the system is running later, the run-time code generator uses the idle cycles of the machine to perpetually *integrate* all components loaded at that moment, recompiling the already executing code base again and again in the background. During these integration cycles, it employs code optimizations that transcend component boundaries. For example, it attempts to optimize cache alignments and inter-procedural register allocation for components that often call each other, based on run-time profiling data. It is also able to in-line procedure calls across component boundaries.

Since these optimizations occur in the background while an alternate version of the same software is already executing in the foreground, the speed of re-compilation is not critical. This means that far more aggressive optimization strategies can be employed than would be possible in an interactive context. As soon as re-compilation is complete, the newly generated code image is substituted for the previous one and re-compilation commences again. Since run-time profiling data is used during re-compilation, successive iterations will yield better and better code.

Hence, iterative dynamic re-compilation can provide the run-time efficiency of a globally optimized monolithic application in the context of dynamically configurable software components. This leads to a new execution model that combines the advantages of conventional application programs with those of modular software (Figure 2). We call this model *quasi-monolithic*, since it exhibits most of the characteristics of a monolithic application. Unlike monolithic applications, however, a quasi-monolithic executable is extensible and can be augmented by further components that communicate with the monolithic core. Over time, the components “outside” of the monolithic core will get drawn inside it, as the run-time code generator integrates them on successive iteration cycles.

component-oriented system with quasi-monolithic core

programming model:	extension of a component hierarchy (components are shared between applications, similar to modular systems, but with more variation in granularity)
distribution model:	independent components that interact with other components and with a shared library base
execution model:	*“quasi-monolithic”* newly arriving components are separate from the already running system for a while, and run less effectively; after a while, they automatically become absorbed by the system and integrated into its code image; periodically, the whole system is re-optimized to reflect recent usage patterns

Figure 2: Programming, Distribution and Execution Models of the Implemented Software Architecture

3 Summary and Conclusion

The quasi-monolithic execution model reconciles modularity and dynamic extensibility with run-time performance. Architectures that are based on this model are well-suited for component-oriented programming.

References

1. Wirth, N. and Gutknecht, J: “The Oberon System”; *Software-Practice and Experience*, 19:9, 857-893; 1989.
2. Wirth, N: “The Programming Language Oberon”; *Software-Practice and Experience*, 18:7, 671-690; 1988.
3. Franz, M: *Code-Generation On-the-Fly: A Key to Portable Software* (Doctoral Dissertation, ETH Zürich); Verlag der Fachvereine, Zürich; 1994.
4. Franz, M. and Kistler, T: *Slim Binaries*; Technical Report No. 96-24, Department of Information and Computer Science, University of California, Irvine; 1996.
5. McIlroy, M. D: “Mass Produced Software Components”; in *Software Engineering, Concepts and Techniques*, Proceedings of the NATO Conferences, New York, 88-98; 1976.