An Strategy to Develop Component Frameworks for Distributed Computing

— extended abstract —

Antônio Augusto Fröhlich
Fraunhofer FIRST
kekuléstraße 7
12489 Berlin, Germany
guto@first.fhg.de
http://www.first.fhg.de/∼guto

1 Introduction

Historically, operating systems have been constructed aiming at abstracting physical resources in a way that is convenient to the hardware, not to applications. Undoubtedly, the monolithic structure of early operating systems contributed to this scenario, for it must have been very difficult, if not impracticable, to customize such systems in order to accomplish the demands of particular applications. The notion that applications have to be adapted to the operating system was so established. Since then, a succession of standardizations has been freezing application program interfaces (API), thus helping to consolidate the situation. Consequently, contemporary operating systems are suffocated by thick layers of standards that prevent internal improvements from reaching applications [Pik00].

Besides failing to accompany the natural evolution of applications, many operating systems also fail to keep updated as regards software engineering. Perhaps this is also a consequence of extreme standardization, whereas there is little room for novel software engineering concepts in the constrained realm of operating systems. Astonishingly, this is a very complex software domain, spanning from hardware to applications, and would greatly profit from modern software engineering techniques. In reality, however, the obsolescence of the techniques deployed in some systems comes out to impact applications.

Even modern systems that support customization have difficulties to match up with application requirements. Mainly because they usually target the design of configurable features, the heart of any customization strategy, on standard compliance and on hardware aspects, and do not adequately address application requirements. Hence, an application programmer may be invited to select features such as POSIX or TCP/IP compliance, or to select drivers for a certain hardware device, but seldom will have the chance to select a distributed object infrastructure. Deploying a general-purpose operating system to support distributed applications is likely to result in a situation where applications get uncountable services that are not needed, but still have to implement much of what is needed. This also leads to the phenomenon that transforms standard APIs, such as CORBA, in middleware layers. A properly constructed run-time support system could deliver its services under a variety of APIs, eliminating such middleware layers.

Building a system as an aggregate of reusable components has the potentiality to considerably improve the case for applications. Nevertheless, component-based software engineering is just a means to construct systems that can be customized to fulfill the demands of particular applications. Inadequately modeled components, or inadequate mechanisms to select and combine components, may render the extra effort of building reusable software components unproductive. The goal of application-driven customization can only be achieved if the system as a whole is designed considering the fulfillment of application require-

ments.

Furthermore, the way customization is typically carried out in component-based systems makes it difficult to pair with application requirements. As a rule, customization in these systems is delegated to end users, which are assisted by some sort of tool in selecting and combining components to produce an executable system. In this case, successfully customizing the system becomes conditioned to the knowledge the user has about it. Hence, user-driven customization is entangled in the balance of component granularity:

- If components are *coarse-grained*, the chance of an ordinary user, i.e., a user without deep knowledge about the system, to successfully conduct customizations grows, but the probability that components will meet application requirements decreases proportionally.
- If components are *fine-grained*, the chance that the system will match application requirement grows, but it is likely that users will not be able to understand the peculiarities of such a large collection of components, probably missing the most adequate configurations.

Improvements in user-driven configuration have been pursued by enabling components to be selected indirectly. The LINUX system, for instance, utilizes a mechanism to select kernel components through the features they implement. Instead of pointing out which components will be included in the system, users can select the desired system features. Features, in turn, are interrelated by dependencies and mapped into components. Nevertheless, even if LINUX kernel components are coarse-grained (they are mainly device drivers and subsystems) and will seldom satisfy the specific requirements of individual applications, selecting features from a list with approximately 700 options[1] is a sordid activity. A mechanism that allows applications themselves to guide the configuration process would be more appropriate.

Notwithstanding, software engineering seems to be mature enough to produce run-time support systems that, besides scaling with the hardware, also scale with applications; that deliver all the functionality required by applications in a form that is convenient for them; and that deduce application requirements to automatically configure itself. Many of the related issues have already been addressed in the context of all-purpose computing by *reflective systems*. In order to comply with the requirements of high-performance distributed applications, the subject is approached in this paper from the perspective of statically configurable component-based systems. The paper elaborates on the *Application-Oriented System Design* method [Frö01] to delineate *an strategy to develop component frameworks for distributed computing*.

2 Domain Engineering

Although no design can go further than its perception of the corresponding problem domain, domain analysis and run-time support systems are subjects that seldom come together. The fact that the operating system domain is basically made of conventions, many of which established long ago in projects such as THE [Dij68] and MULTICS [Org72], seems to have fastened it to a "canonical" partitioning. This partitioning includes abstractions such as process, file, and semaphore, and is taken "as-is" by most designers. Indeed, it is now consolidated by standards on one side and by the hardware on the other, leaving very little room for new interpretations.

Notwithstanding this, revisiting the problem domain during the design of a new operating system would probably reveal abstractions that are better tuned with contemporary applications. For example, the triple (process, file, message passing) could be replaced by persistent communicating active objects. Actually, most run-time platforms feature this perspective of the operating system domain through a middleware layer such as CORBA [OMG01] and JAVA [SUN01]. However, the middleware approach goes the opposite direction of application-orientation, whereas it further generalizes an already generic system.

Nevertheless, even if one endures domain analysis knowing that decomposition will have to be carried out respecting the boundaries dictated by standards, programming languages, and hardware, there is at least

[1] The number of LINUX configurable kernel features has been estimated by executing the following command in a system based on kernel version 2.2.14: `grep CONFIG /usr/src/linux/configs/kernel-2.2.14-i386.config | wc -l`.

one important reason to do it: to avoid the monolithic representation of abstractions. If an application-oriented operating system is to be the output of design, capturing application-specific perspectives of each abstraction and modeling them as independently deployable units, as suggested by *subject-oriented programming* [HO93], is far more adequate than the monolithic approach. After all, the product of domain engineering is not a single system, but a collection of reusable software artifacts that model domain entities and can be used to build several systems.

2.1 Application-Oriented Domain Decomposition

An application-oriented decomposition of the problem domain can be obtained, in principle, following the guidelines of *object-oriented decomposition* [Boo94]. However, some subtle yet important differences must be considered. First, object-oriented decomposition gathers objects with similar behavior in class hierarchies by applying variability analysis to identify how one entity specializes the other. Besides leading to the infamous "fragile base class" problem [MS98], this policy assumes that specializations of an abstraction (i.e. *subclasses*) are only deployed in presence of their more generic versions (i.e. *superclasses*).

Applying variability analysis in the sense of *family-based design* [Par76] to produce independently deployable abstractions, modeled as members of a family, can avoid this restriction and improve on application-orientation. Certainly, some family members will still be modeled as specializations of others, as in *incremental system design* [HFC76], but this is no longer an imperative rule. For example, instead of modeling connection-oriented as a specialization of connectionless communication (or vice-versa), what would misuse a network that natively operates in the opposite mode, one could model both as autonomous members of a family.

A second important difference between application-oriented and object-oriented decomposition concerns environmental dependencies. Variability analysis, as carried out in object-oriented decomposition, does not emphasizes the differentiation of variations that belong to the essence of an abstraction from those that emanate from execution scenarios being considered for it. Abstractions that incorporate environmental dependencies have a smaller chance of being reused in new scenarios, and, given that an application-oriented system will be confronted with a new scenario virtually every time a new application is defined, allowing such dependencies could severely hamper the system.

Nevertheless, one can reduce such dependencies by applying the key concept of *aspect-oriented programming* [KLM+97], i.e. aspect separation, to the decomposition process. By doing so, one can tell variations that will shape new family members from those that will yield scenario aspects. For example, instead of modeling a new member for a family of communication mechanisms that is able to operate in the presence of multiple threads, one could model multithreading as a scenario aspect that, when activated, would lock the communication mechanism (or some of its operations) in a critical section.

The phenomenon of mixing scenario aspects and abstractions seems to happen spontaneously in most other design methods, thus learning to avoid it might require some practice. Perhaps the most critical point is the fact that systems are often conceived with an implementation platform in mind, which is often better understood than the corresponding problem domain. In principle, there is nothing wrong in studying the target platform in details before designing a system, actually it might considerably save time, but designers tend to misrepresent abstractions while considering how they will be implemented in the chosen platform. In an application-oriented system design, this knowledge about implementation details should be driven to identify and isolate scenario aspects. In general, aspects such as identification, sharing, synchronization, remote invocation, authentication, access control, encryption, profiling, and debugging can be represented as scenario aspects.

Building families of scenario-independent abstractions and identifying scenario aspects are the main activities in application-oriented domain decomposition, but certainly not the only ones. The primary strategy to add functionality to a family of abstractions is the definition of new members, but sometimes it is desirable to extend the behavior of all members at once. Specializing each member would double the cardinality of the family. Application-oriented system design deals with cases like this by modeling the extended func-

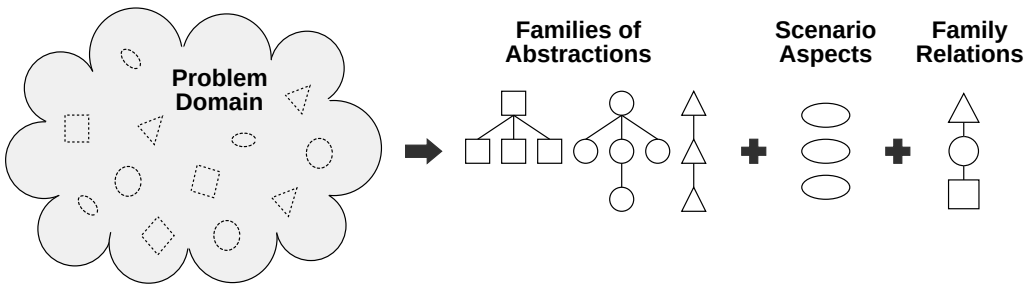

Figure 1: An overview of application-oriented domain decomposition.

tionality as a *configurable feature*. Just like scenario aspects, configurable features modify the behavior of all members of a family when activated, but, unlike those, are not transparent. One could say that scenario aspects have "push" semantics, while configurable features have "pull".

A configurable feature encapsulates common data structures and algorithms that are useful to implement a family's feature, but leave the actual implementation up to each family member. Abstractions are free to reuse, extend, or override what is provided in a configurable feature, but are requested to behave accordingly when the feature is enabled.

The case for configurable features can be illustrated with a family of networks and features such as multicasting, in-order delivery, and error detection. If new family members were to be modeled for each such a feature, a family of 10 networks subjected to 10 features could grow up to 10^{10} members. Modeling this kind of feature as a scenario aspect is usually not possible either, since its implementation would have to be specialized to consider particular network architectures.

Another relevant issue to be considered during domain decomposition is how abstractions of different families interact. Capturing ad-hoc relationships between families during design can be useful to model reusable software architectures, helping to solve one of the biggest problems in component-based software engineering: how to tell correct, meaningful component compositions from unusable ones. A reusable architecture avoids this question by only allowing predefined compositions to be carried out. For example, one could determine that the members of a family of process abstractions must use the family of memory to load code and data, avoiding an erroneous composition with members of the file family. In application-oriented system design, reusable architectures are captured in component frameworks that define how abstractions of distinct families interact. Although such frameworks are defined much later in the design process, taking note of ad-hoc relationships during domain decomposition can considerably ease that activity.

An overview of application-oriented domain decomposition is presented in figure 1. In summary, it is a multiparadigm domain engineering method that promotes the construction of application-oriented systems by decomposing the corresponding domain in families of reusable, scenario-independent abstractions and the respective scenario aspects. Reusable system architectures are envisioned by the identification of inter-family relationships that will later build component frameworks.

The full version of this paper will elaborate on these concepts to delineate a systematic strategy to develop component frameworks for distributed computing, including an analysis of distributed computing domain and an overview of EPOS, an application-oriented operating system that promptly serves applications in this domain.

References

[Boo94] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 2 edition, 1994.

[Dij68] Edsger Wybe Dijkstra. The Structure of the THE-Multiprogramming System. *Communications of the ACM*, 11(5):341–346, May 1968.

[Frö01] Antônio Augusto Fröhlich. *Application-Oriented Operating Systems*. Number 17 in GMD Research Series. GMD - Forschungszentrum Informationstechnik, Sankt Augustin, August 2001.

[HFC76] A. Nico Habermann, Lawrence Flon, and Lee W. Cooprider. Modularization and Hierarchy in a Family of Operating Systems. *Communications of the ACM*, 19(5):266–272, 1976.

[HO93] William H. Harrison and Harold Ossher. Subject-oriented Programming (a Critique of Pure Objects). In *In Proceedings of the 8th Conference on Object-oriented Programming Systems, Languages and Applications*, pages 411–428, Washington, U.S.A., September 1993.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-oriented Programming'97*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, June 1997. Springer.

[MS98] Leonid Mikhajlov and Emil Sekerinski. A Study of the Fragile Base Class Problem. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 355–382, Brussels, Belgium, July 1998. Springer.

[OMG01] Object Management Group. *Common Object Request Broker Architecture*, online edition, January 2001. [http://www.corba.org/].

[Org72] Elliott Organick. *The Multics System: an Examination of its Structure*. MIT Press, Cambridge, U.S.A., 1972.

[Par76] David Lorge Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, March 1976.

[Pik00] Rob Pike. Systems Software Research is Irrelevant. Online, February 2000. [http://cm.bell-labs.com/who/rob/utah2000.ps].

[SUN01] SUN Microsystems. *The Source for Java Technology*, online edition, January 2001. [http://java.sun.com/].