A Strategy to Develop Component Frameworks for Parallel Computing

Abstract

Properly bridging a parallel architecture to a given class of parallel applications is a defying task that calls for custom designed run-time support systems. *Only in this way would applications get minimal-overhead system services delivered through application-oriented interfaces. Designing and implementing a new run-time support system for each class of parallel applications, however, would be impracticable in terms of costs. Notwithstanding, software component engineering could help to meet a compromise with applications by means of tailorable run-time systems. This paper describes a strategy to develop component frameworks for parallel computing based on the Application-Oriented System Design method proposed by Fröhlich [3]. These frameworks are the substrata to build tailorable run-time support systems.*

Keywords: *application-oriented system design, component frameworks, aspect-oriented programming, static metaprogramming, generative programming.*

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 [13].

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 realm, that spans 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 parallel object infrastructure. Deploying a general-purpose operating system to support parallel 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 MPI, 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 *software 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 requirements.

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 though 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 1450 options[1] is

[1]The number of LINUX configurable kernel features has been estimated by executing the following command in a system based on kernel version 2.4.18: `grep CONFIG /usr/src/linux/.config | wc -l`.

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 parallel 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 [3] to delineate *a strategy to develop component frameworks for parallel computing*.

2. Domain Engineering

Although no design can go further than its perception of the corresponding problem domain, domain engineering 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 [2] and MULTICS [11], 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, socket)` 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 [10] and JAVA [15]. 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 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* [7], 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* [1]. 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 [9], 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* [12] 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* [6], 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* [8], 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 functionality 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

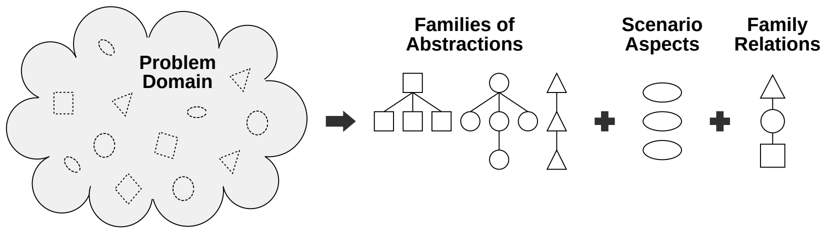

Figure 1. An overview of application-oriented domain decomposition.

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 multi-paradigm 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.

3. Component Frameworks: capturing reusable system architectures

Along with the specification of abstraction families and scenario aspects, an application-oriented system design delivers specifications of reusable system architectures, which define how abstractions can be arranged together in a functioning system. Reusable system architectures are usually defined considering the domain knowledge acquired during the design of previous systems. After having developed some systems, or some versions of a system, for a certain problem domain, developers begin to agree on how to implement the abstractions that build up the domain; how they interact with each other, with the environment, and with applications; and how the implied non-functional requirements can be accomplished. Such an expertise can be captured in an architectural specification to be reused in upcoming systems.

Capturing reusable system architectures in a component-based system is fundamental, since a pile of components, by itself, is nothing but a pile of components. A component-based system is only achieved when components can be arranged together in an assemblage of predictable behavior. In application-oriented system design, reusable architectures begin to be modeled yet during domain decomposition with the identification of relationships between families of abstractions. These relationships are enriched by scenario constraints during the specification of scenario aspects and serve as input for this phase, which aims at producing a detailed specification of reusable system architectures in the form of component frameworks.

An *application-oriented component framework* captures a reusable architecture by specifying the families of abstractions that take part in a certain kind of system, as well as rules that guide their interaction. Systems produced by component frameworks, when compared to arbitrary arrangements of components, are less prone to misbehavior, since only compositions that have been envisioned by system architects are allowed. Although component frameworks are not the unique alternative to capture reusable architectures—among others, *aspect programs*, *subjects*, and *collaborations* could also be used for this purpose—they fit perfectly with application-orientation's notion of isolating scenario aspects from abstractions by means of scenario adapters [5].

An application-oriented component framework could be defined as a collection of interrelated scenario adapters as shown in figure 2. Each scenario adapter would set up a "socket" for components of the corresponding family. Plugging components into the framework would be accomplished by binding the inflated interface of every used family to the desired family member. The way scenario adapters are arranged in the framework would define the basic architecture of resultant systems, while architectural elements that do not concern components could be hard-coded in the framework.

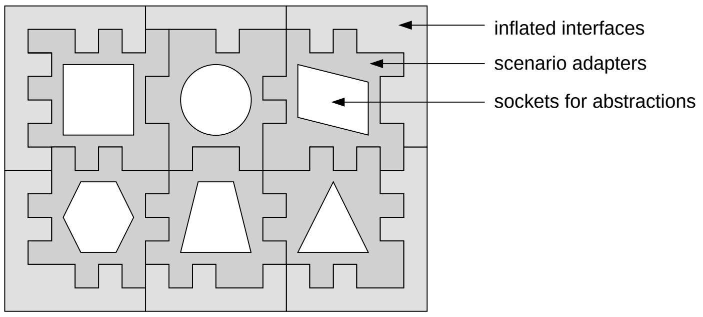

Figure 2. An application-oriented component framework.

A component framework defined in terms of scenario adapters would also present advantages concerning *system-wide properties*, which could be modeled as scenario aspects to be enforced on components by the respective scenario adapters [16]. Moreover, a component framework of this kind does not require complex code manipulation tools to generate a system instance. After all, the representation of a component framework as a socket board to which components can be plugged is well understood and accepted by users.

Figure 3 shows a schematic representation of a component framework that embodies a plausible system architecture for the domain of high-performance communication in clusters of workstations. It illustrates the relationships between three families of abstractions: `Communication End-Point`, `Communication Strategy`, and `Network`. Firstly, it shows a mutual dependency between the families of strategies and end-points, i.e., by selecting a certain strategy, one automatically selects the corresponding end-point, and vice-versa[2]. It also shows that a network is *used* by the members of the communication strategy family. The respective sets of components are also shown to illustrate the "select-and-plug" organization of the framework.

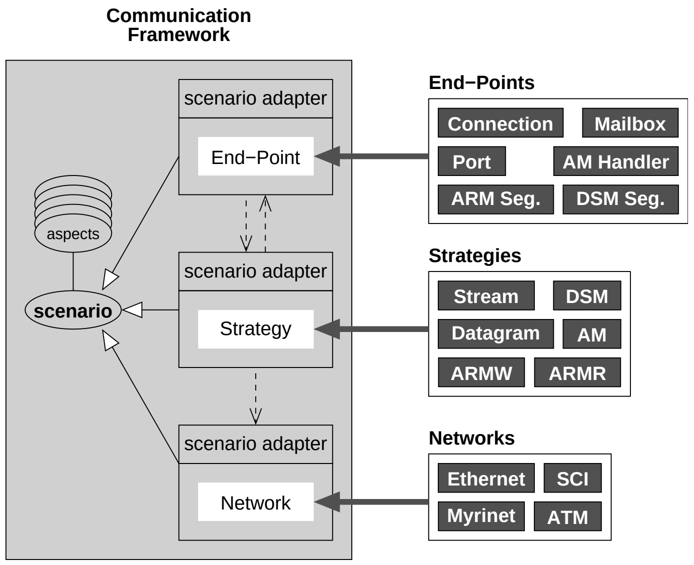

Figure 3. A component framework for the domain of high-performance communication.

Arbitrary compositions handled in the real of this component framework could be adapted to scenarios such as *thread safe*, *buffered*, and *reliable* simply by selecting the proper scenario aspects.

4. Case Study: EPOS component framework

The EPOS system was born in 1997 at the Research Institute for Computer Architecture and Software Engineering (FIRST) of the German National Research Center for Information Technology (GMD) as a project to experiment with the concepts and mechanisms of application-oriented system design. Indeed, EPOS and application-oriented system design cannot be disassociated, since both have been evolving together from the very beginning.

The acronym EPOS stands for *Embedded Parallel Operating System*. It was coined considering the main research goal established for the early system: to embed the operating system in a parallel application. The strategy followed to achieve this goal consisted in modeling the corresponding domain entities as a set of reusable and adaptable components, and developing a mechanism to allow parallel applications to easily specify constraints to guide the arrangement of these components in a functioning system. As the system evolved, it became clear that this strategy could be applied

[2]In this model, `Connection` is the end-point for the `Stream` strategy, `Port` and `Mailbox` for `Datagram`, `Active Message Handler` for `Active Message`, `Asynchronous Remote Memory Segment` for `Asynchornous Remote Memory Read/Write/Copy`, and `Dsitributed Shared Memory Segment` is the end-point for the `Distributed Shared Memory` strategy.

to a broader universe of applications. Concerning design and organization, EPOS is inherently tied with dedicated computing and static configurability, but whether a platform is dedicated to an application temporarily (like in traditional parallel systems) or permanently (like in most embedded systems) does not play a significant role. Hence, *Embedded Parallel Operating System* can also be interpreted as a system that targets both embedded and parallel applications.

EPOS owes much of its philosophy to the PEACE system [14], from which it inherited the notion that "generic" and "optimal" are adjectives that cannot be simultaneously applied to the same operating system, besides a rich perception of family-based design. EPOS implementation for the Intel iX86 architecture reuses some of the strategies adopted in ABOELHA [4], a former research operating system developed by the author. However, the design of ABOELHA did not promote reuse, so these strategies had to be completely remodeled for EPOS.

EPOS component framework is realized by a *static metaprogram* and a set of *composition rules*. The metaprogram is responsible for adapting system abstractions to the selected execution scenario and arranging them together during the compilation of an application-oriented version of EPOS. Rules coordinate the operation of the metaprogram, specifying constraints and dependencies for the composition of system abstractions. Composition rules are not encoded in the metaprogram, but specified externally. They are interpreted by composition tools in order to adjust the parameters of the metaprogram.

The separation of composition rules from the framework metaprogram allows a single framework to yield a variety of software architectures. Indeed, one could say that EPOS has many frameworks, each corresponding to the execution of the metaprogram with a different set of arguments. Moreover, the use of static metaprogramming to compose system abstractions does not incur in run-time overhead, thus yielding composites whose performance is directly derived from their parts.

4.1. Framework Metaprogram

EPOS *component framework metaprogram* is executed in the course of a system instance compilation, adapting selected abstractions to coexist with each other and with applications in the designated execution scenario. During this process, scenario-independent abstractions have their original properties preserved, so that internal compositions can be carried out before scenario adaptation. This is accomplished having the framework metaprogram to import scenario-independent abstractions in one namespace and export the corresponding scenario-adapted versions in another.

For example, the cascaded aggregation of `Communication End-Point`, `Communication Strategy`, and `Network` illustrated in the previous section would take place at the scenario-independent level. The resultant composite would later be adapted to the selected scenario as a whole.

Figure 4 shows a top-view diagram of the component framework static metaprogram. Each of the elements represented in the figure will be subsequently described. A class diagram representing the `Handle` framework element is depicted in figure 5. Like most other elements, parameterized class `Handle` takes a system abstraction (class of system objects) as parameter. When instantiated, it acts as a "handle" for the supplied abstraction, realizing its interface in order that invocations of its methods are forwarded to `Stub`. Hence, system objects are manipulated by applications via their "handles".

`Handle` provides additional operations to check if a system object was successfully created[3] and to obtain its id.

[3]The use of C++ exceptions as a mechanism to signalize system object creation failure was avoided because it would make difficult the

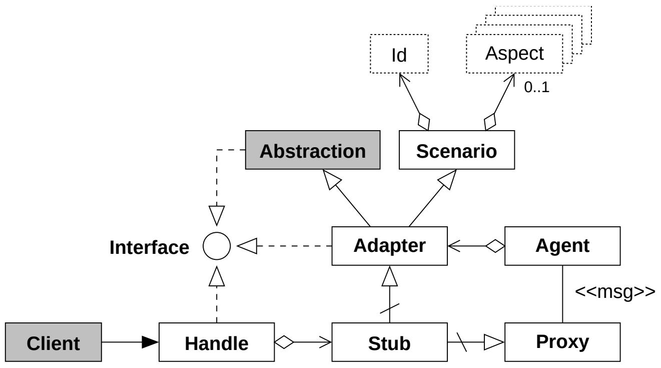

Figure 4. A top-view of EPOS component framework metaprogram.

Besides, when the `Shared` scenario aspect is enabled, `Handle` provides the extra constructors used to designate sharing in that scenario. The aggregation relationship between `Handle` and `Stub` enables the system to enforce allocation via a system allocator (instead of a programming language one), thus allowing for the `Allocated` scenario aspect[4].

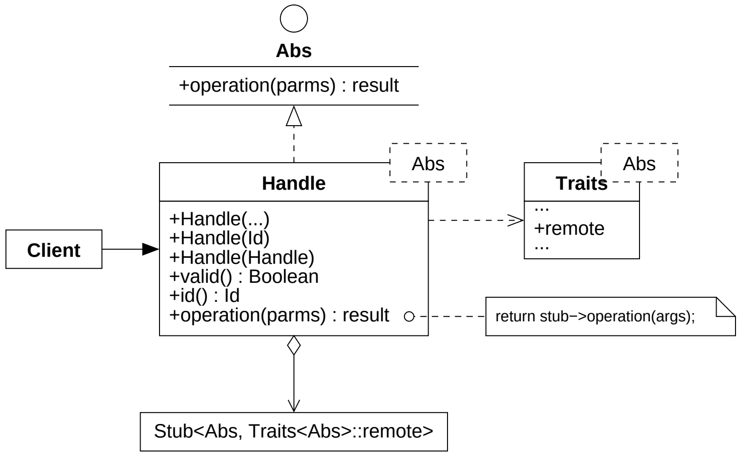
Abs
+operation(parms) : result
Abs
Handle
+Handle(...)
+Handle(Id)
+Handle(Handle)
+valid() : Boolean
+id() : Id
+operation(parms) : result
Abs
Traits
...
+remote
...
Client
return stub−>operation(args);
Stub<Abs, Traits<Abs>::remote>

Figure 5. EPOS framework: the `Handle` element.

The `Stub` framework element is depicted in figure 6. This parameterized class is responsible for bridging `Handle` either with the abstraction's scenario adapter or with its proxy. It declares two formal parameters: an abstraction and a boolean flag that designates whether the abstraction is local or remote to the address space of the calling process. By default, `Stub` inherits the abstraction's scenario adapter, but it has a specialization, namely `Stub<Abs, true>`, that inherits the abstraction's proxy. Therefore, making `Traits<Abstraction>::remote = false` causes `Handle` to take the scenario adapter as the `Stub`, while making it `true` causes `Handle` to take the proxy.

The `Proxy` framework element is deployed in a remote invocation scenario. `Proxy` realizes the interface of the abstraction it represents, forwarding method invocations to its `Agent` (see figure 7). Each instance of `Proxy` has a

integration of EPOS with applications written in other programming languages.

[4]The allocator used with each abstraction is selected by `Adapter` after consulting `Traits<Abs>::allocated`.

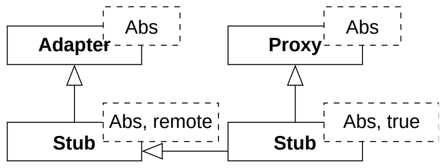

Figure 6. EPOS **framework: the** `Stub` **element.**

private ROI message, which is initialized in such a way that forthcoming invocations only need to push parameters into the message. Moreover, because `Proxy` is metaprogrammed, parameters are pushed directly into the message, without being pushed into the stack first. `Proxy` operations invoke method `invoke`[5] to perform a message exchange with `Agent`, which, likewise `Handle` for a local scenario, forwards invocations to the abstraction's `Adapter`.

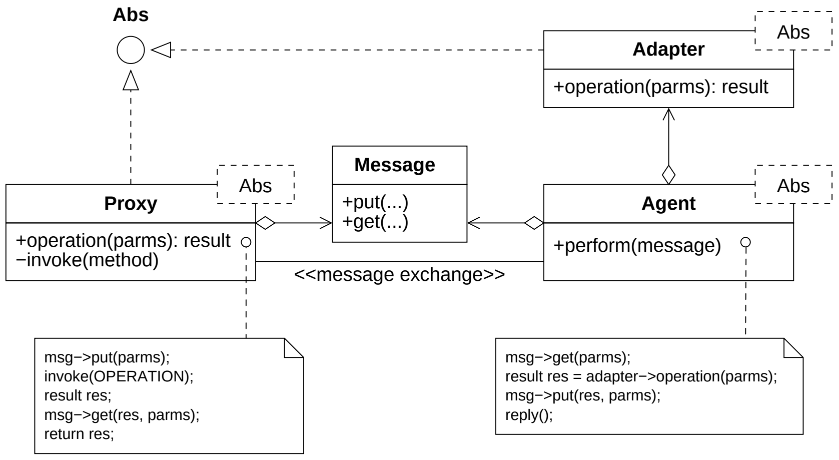

Figure 7. EPOS **framework:** `Proxy` **and** `Agent` **elements.**

The `Adapter` framework element is depicted in the figure 8. This parameterized class realizes a scenario adapter for the abstraction it takes as parameter, adapting its instances to perform in the selected scenario. Adaptations are carried out by wrapping the operations defined by the abstraction within the `enter` and `leave` scenario primitives, and also by enforcing a scenario-specific semantics for creating, sharing, and destroying its instances. The role of `Adapter` in EPOS framework is to apply the primitives supplied by `Scenario` to abstractions, without making assumptions about the scenario aspects represented in these primitives. In this way, `Adapter` is able to enforce any combination of scenario aspects.

The execution scenario for EPOS abstractions is ultimately shaped by the `Scenario` framework element depicted in figure 9. Each instance of parameterized class `Scenario` delivers scenario primitives that are specific to the abstraction supplied as parameter. Firstly, it incorporates the selected `Id` aspect, which is common to all abstractions in a scenario; then it consults the abstraction's `Traits` to determine which aspects apply to it, aggregating the corresponding scenario aspects. The strategy to cancel an aggregation is similar to the one used with `Stub`, i.e. a parameterized class that inherits the selected aspect by default, but is specialized to inherit nothing in case the aspect

[5]The semantics of the `invoke` method varies according to the selected configuration. In some cases, it causes the application process to "trap" into the kernel, in others, it directly accesses a communicator to perform a message exchange.

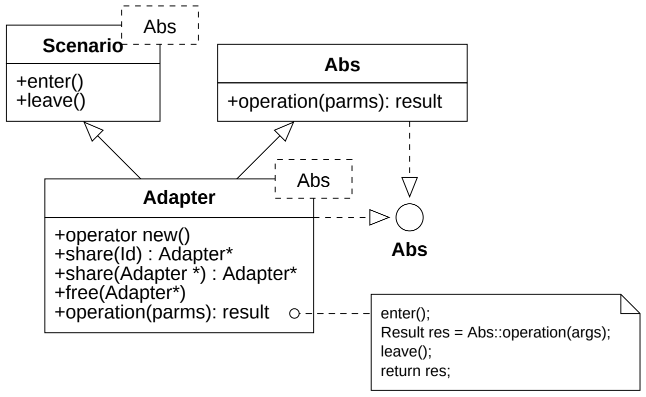

Figure 8. EPOS **framework: the** `Adapter` **element.**

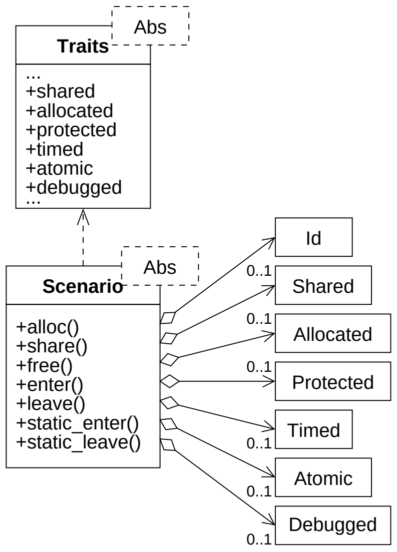

Figure 9. EPOS **framework: the** `Scenario` **element.**

is not selected for the abstraction.

Besides designating which scenario aspects apply to each abstraction, the parameterized class `Traits` maintains a comprehensive compile-time description of abstractions that is used by the metaprogram whenever an abstraction-specific element has to be configured.

4.2. Composition Rules

EPOS component framework metaprogram is able to adapt and assemble selected components to produce an application-oriented operating system. However, though the metaprogram knows about particular characteristics of each system abstraction from its *traits*, it does not know of relationships between abstractions and hence cannot guarantee the consistency of the composites it produces. In order to generate a meaningful instance of EPOS, the metaprogram must be invoked with a coherent parameter configuration.

Therefore, the operation of the framework metaprogram is coordinated by a set of *composition rules* that express elements of reusable system architecture captured during design. A consistent instance of EPOS comprises system abstractions, scenario aspects, hardware mediators, configurable features, and non-functional requirements. Composition rules specify dependencies and constraints on such elements, so that invalid configurations can be detected and rejected. Nevertheless, guarantying that a composite of EPOS elements is “correct” would depend on the formal specification and validation of each element, what is outside the scope of this research.

Sometimes, composition rules are implicitly expressed during the implementation of components. For example, by referring to the `Datagram` channel, the `Port` communicator implicitly specifies a dependency rule that requires `Datagram` to be included in the configuration whenever `Port` is deployed. However, most composition rules, especially those designating constraints on combining abstractions, can only be expressed externally. For instance, the rule that expresses the inability of the `Flat` address space to support the `Mutual` task abstraction must be explicitly written.

In order to support the external specification of composition rules, EPOS elements are tagged with a *configuration key*. When a key is asserted, the corresponding element is included in the configuration. Elements that are organized in families are selected by assigning a member’s key to the family’s key, causing the family’s inflated interface to be bound to the designated realization. This mechanism implements the selective realize relationships modeled during design. For example, writing `Synchronizer := Semaphore` causes the inflated interface of the `Synchronizer` family of abstractions to be bound to member `Semaphore` and writing `Id := Capability` binds the `Id` scenario aspect to `Capability`. Elements that do not belong to families have their keys asserted accordingly. For example, writing `Busy_Waiting := True` enables the `Busy_Waiting` configurable feature if the `CPU_Scheduler` abstraction.

Composition rules are thus defined associating pre- and postconditions to configuration keys. For instance, the following rule for the `Task` family of abstractions requires the `Paged` address space to be selected before the `Mutal` task can be selected:

```
Mutual ⇒ pre:  Address_Space = Paged
```

Alternatively, this constraint could be expressed as a composition rule for the `Address_Space` family that selects the `Exclusive` task whenever the `Flat` address space is selected:

$$\texttt{Flat} \Rightarrow \texttt{pos: Task := Exclusive}$$

Composition rules are intended to be automatically processed by configuration tools. Hence, it is fundamental to keep them free of cycles. The following rule, though understandable for a human, could bring a tool to deadlock:

$$\texttt{A1} \Rightarrow \texttt{pre: B = B1}$$
$$\texttt{B1} \Rightarrow \texttt{pre: A = A1}$$

In order to ensure that EPOS composition rules build a direct acyclic graph, the following directives were observed:

- Configuration keys are *totally ordered* according to an arbitrary criterion;
- Preconditions are restricted to *expressions* involving only *preceding* keys;
- Postconditions are restricted to *assignments* involving only *succeeding* keys.

Along with the *traits* of each abstraction, composition rules control the process of tailoring EPOS to a particular application. The set of configuration keys selected by the user is validated and refined by means of composition rules, yielding a set of elements that are subsequently assembled by the framework metaprogram consulting the traits of abstractions.

5. Conclusions

Historically, applications have been adapted to the operating system, adhering to standardized application program interfaces that covey uncountable useless services (for each individual application), and yet fail to deliver much of what is necessary. An application-oriented operating system ruptures with this notion, implementing services that emanate from application requirements and delivering them as a set of configurable components that can be assembled to produce application-tailored system instances.

In this paper, a strategy to deploy the application-oriented system design method in order to build component frameworks for parallel computing has been proposed. Frameworks produced under this strategy consist of intertwined scenario adapters that build a socket board for software components. The case study on EPOS component framework illustrates the strategy, supplying run-time system designers with valuable guidelines to meet the parallel application challenge.

References

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

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

[3] A. A. Fröhlich. *Application-Oriented Operating Systems*. Number 17 in GMD Research Series. GMD - Forschungszentrum Informationstechnik, Sankt Augustin, Aug. 2001.

[4] A. A. Fröhlich, R. B. Avila, L. Piccoli, and H. Savietto. A Concurrent Programming Environment for the i486. In *Proceedings of the 5th International Conference on Information Systems Analysis and Synthesis*, Orlando, USA, July 1996.

[5] A. A. Fröhlich and W. Schröder-Preikschat. Scenario Adapters: Efficiently Adapting Components. In *Proceedings of the 4th World Multiconference on Systemics, Cybernetics and Informatics*, Orlando, U.S.A., July 2000.

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

[7] W. H. Harrison and H. 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., Sept. 1993.

[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. 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.

[9] L. Mikhajlov and E. 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.

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

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

[12] D. L. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, Mar. 1976.

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

[14] W. Schröder-Preikschat. PEACE - A Software Backplane for Parallel Computing. *Parallel Computing*, 20(10):1471–1485, 1994.

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

[16] C. Szyperski and R. Vernik. Establishing System-Wide Properties of Component-Based Systems: A Case for Tiered Component Frameworks. In *In Proceedings of the Workshop on Compositional Software Architectures*, Monterey, U.S.A., Jan. 1998.