# A Framework for Developing Configurable Objects

Dilma M. Silva
Computer Science Department
University of São Paulo
Rua do Matão, 1010
05508-900 São Paulo, Brazil
dilma@ime.usp.br

Karsten Schwan
College of Computing
Georgia Institute of Technlogy
Atlanta, GA 30332
schwan@cc.gatech.edu

## Resumo

A crescente importância de novas áreas de aplicação como sistemas multimídia, sistemas colaborativos e distribuição de informação na Internet vem aumentando a demanda por flexibilidade no software. Este artigo apresenta um arcabouço (COBS$^{OM}$) para a construção de programas paralelos e distribuídos configuráveis onde a funcionalidade associada ao tipo de um objeto é explicitamente separada de outras características como desempenho, confiabilidade e propriedades temporais. COBS$^{OM}$ sustenta um modelo de programação onde o manuseio da configuração é uma aspecto central no projeto, provendo abstrações para incorporar flexibilidade em um sistema orientado a objetos de forma metódica. Além disto, aspectos de desempenho são explorados através de ajuste dinâmicos (em tempo de execução) dos mecanismos que os influenciem. Apresentamos os elementos básicos de nosso modelo de configuração, assim como Data_Object, um objeto configurável complexo que encapsula os dados de saída de uma aplicação científica paralela e distribuída de alto desempenho.
**Palavras-chave:** configuração, flexibilidade, projeto orientado a objetos, objetos distribuídos

## Abstract

The recent boom of new application categories, such as multi-media systems, groupware, and the wide area distribution of information across the Internet, has led to further demands for flexibility in software. This paper presents a framework (COBS$^{OM}$) for building configurable parallel and distributed programs where type-dependent object functionality is explicitly separated from its characteristics subject to configuration, including its performance, reliability, and timing properties. COBS$^{OM}$ supports a programming model where dealing with configuration issues is a central part of the design. It provides abstractions for incorporating flexibility into a distributed object-oriented application in a methodical fashion. In addition, performance issues are addressed by considering run-time execution adjustments of the basic mechanisms that influence them. We introduce the basic elements of the model. We also present Data_Object, which has been developed with COBS$^{OM}$. Data_Object is a complex configurable object that encapsulates data output from a high performance parallel and distributed scientific application.
**Keywords:** configuration, flexibility, object design,distributed objects

## 1 Introduction

*Software flexibility* is an important issue during the development of high performance and real-time applications, reliable systems, and in exploratory computing[27]. Furthermore, flexibility is perceived as a generally desirable software characteristic, since it facilitates the adaptation of a software product to new execution environments, usage constraints, and functionality requirements. The recent boom of new application categories, such as multi-media systems and the wide area distribution of information across the Internet, has led to further demands for flexibility in software. Namely, in all such applications, the attainment of reasonable levels of performance requires the exploitation of specific characteristics of their execution environments, typically by the execution of behaviors specialized for them. As a consequence, the software development process should address *runtime flexibility* as a crucial requirement for current and emerging application domains by incorporating adaptation capabilities into software components. Specifically, we aim to offer *configurable objects* as a means of achieving flexible systems in which runtime execution adjustments lead to improved performance. Namely, a flexible software element should be able to adjust itself to its current execution environment in such a way that it can mimic the performance of an object customized for the environment. This not only requires the element to be configurable, but also capable of understanding its execution environment and its relationship with other software components.

Our work explores configurability issues. The goal of this work is the development of programming environment support for reasoning about and dealing with configuration issues. The framework we have constructed, COBS$^{OM}$, (1) addresses performance issues by considering the basic mechanisms that influence them; and (2) provides abstractions for incorporating flexibility into a distributed object program in a methodical fashion.

Runtime adaptation of high performance scientific applications has been investigated by our group for many years[9, 8]. In collaboration with atmospheric scientists at Georgia Tech, we have developed a parallel and distributed global chemical transport model[16] capable of running on any of the high performance engines in our computing environment (Figure 1). Models like this are important tools for answering scientific questions concerning the distribution of chemical species such as chlrofluorocarbons, hydrochlorofluorocarbon, and ozone. This model generates a very large set of data at each time step of the simulation. As a result, making the data available to the end user brings out problems similar to data mining in large database systems. Using COBS$^{OM}$, we developed a configurable object that offers flexible and efficient access to the model's data, as described in Section 4.

## 2 Related Work

Runtime flexibility per se is not a new concept, as evident from the early uses of self modifying code in operating systems. As hardware capabilities evolved, software technology has advanced and it has become possible for program designers to consider a diversity of strategies and paradigms in order to match widely varying application requirements.

Both the high performance computing and the operating system communities have

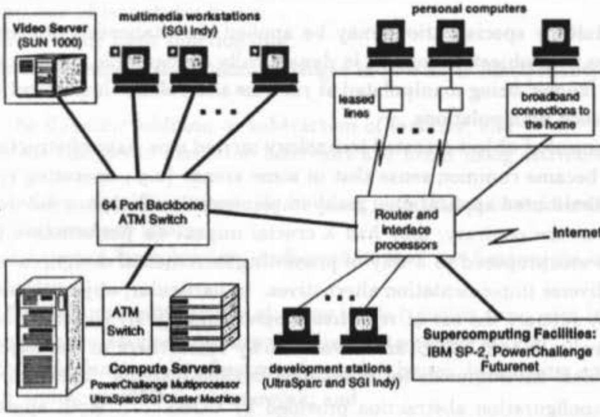
Video Server (SUN 1000)
multimedia workstations (SGI Indy)
personal computers
leased lines
broadband connections to the home
64 Port Backbone ATM Switch
Router and interface processors
Internet
ATM Switch
Compute Servers
PowerChallenge Multiprocessor
UltraSparc/SGI Cluster Machine
development stations (UltraSparc and SGI Indy)
Supercomputing Facilities: IBM SP-2, PowerChallenge Futurenet

Figura 1: The computing environment in the Distributed Laboratories Project at Georgia Tech.

used program configuration to improve performance or reliability. A variety of research results has enabled the runtime configuration of operating systems in order to improve the performance of specific user programs, including the early work on the removal of operating system services from 'fixed' kernels to the configurable user level in the Mach[22] and NT[6] operating systems, the specialization of critical fragments of code in Synthesis[19], and the notion of micro-kernels and user-level libraries for implementing customized operating system abstractions[10]. In effect, such research has established the fact that application programs may be 'combined' with operating system functions such that both may be configured jointly, using the same programming techniques and software infrastructures. Such joint configuration is explored in several recent object-based efforts[14], including the Choices[4], Spring[13], Chaos[24], Apertos[28], and ACE[23] operating systems.

For parallel programming, the chosen levels of parallelism, scheduling, and synchronization mechanisms may vary based on the data and resources available. In the work described in [20] we achieved performance improvements through the dynamic adaptation of object and invocation implementations. We built a *Configuration Toolkit* (CTK)[26], which is a library for constructing configurable object-based abstractions implementing multiprocessor programs or operating system components. The library is unique in its exploration of runtime configuration for attaining performance improvements: (1) its programming model facilitates the expression and implementation of program configuration, and (2) its efficient runtime support enables performance improvements by configuration of program components during their execution. Program configuration is attained without compromising the encapsulation or the reuse of software abstractions, by explicitly separating the type-dependent object functionality from its properties subject to configuration, including its performance, reliability, and timing properties. Using CTK, objects may be specialized using diverse techniques, including parameterization and interposition. Multiple specializations may be applied simultaneously by association of multiple policies with objects, resulting in dynamically configurable systems where attributes resemble 'knobs' being manipulated at runtime and policies implement the changes resulting from such manipulations.

The development of object oriented technology incited new ways of structuring implementations. It became common sense that in some arenas (*e.g.,* operating systems, real time systems, distributed applications) many implementation decisions did not represent "just details"; on the contrary, they had a crucial impact on performance ([14]). Object orientation was proposed as a way of promoting incremental design, robustness and incorporating diverse implementation alternatives. In particular, object oriented languages that directly support the use of *reflective programming* ([17]) and *meta-objects* ([12], [7]) such as Smalltalk and CLOS are advocated by researchers in the object oriented community as ideal environments for developing flexible systems. The *meta-objects* are similar to the configuration abstraction provided by COBS$^{OM}$. Both approaches provide mechanisms for changing an object's behaviors dynamically in most of its aspects (object creation, method invocation, etc), but in general, meta-object protocols address configuration problems for which efficient runtime configuration is not crucial.

In SOM[7] or CLOS[21], reflection principles are used for changing object behavior by invoking the methods available to create and initialize classes, to compose their methods tables, etc. As a result, object descriptions may be altered at runtime by invoking the inherited methods for dealing with class objects, thereby attaining configuration by manipulation of a potentially complex class description. In comparison, object configuration in our work is relatively 'lightweight' since it involves only the manipulation of objects that have been specifically created for purposes of object configuration. This makes our framework more suitable for attaining performance gains via configuration, whereas SOM and CLOS address issues like the evolution of compatible object libraries by configuration of entire object system structures.

The *open implementation* approach was proposed in the context of meta-objects and reflective programming[11], and evolved into the use of those ideas in environments where performance requirements prohibit the maintenance of a complete framework for interpreting and redefining object properties. Recent work on the dynamic configuration of distributed and object-based systems[5] (such as Polylith[1], Regis[18], Equus[15]) often concerns specific configuration methods or general (rather than high performance) frameworks for implementing dynamically configurable applications.

## 3 Objects, Configuration Entities and Configuration Channels

In this work, we assume flexible systems as being composed of abstractions that can be dynamically configured in terms of (1) their implementations, (2) how they use other resources in the system, and (3) their requirements in terms of performance, reliability, or application needs. Such abstractions can be built and tailored for specific needs by connecting a set of objects; some objects encapsulate the desired basic functionality while others carry out the work related to each one of the configurable facets of object behavior.

In other words, our object abstractions:

- encapsulate some basic functionality;
- are able to accommodate dynamic changes in how their functionality is implemented;
- permit the dynamic addition or subtraction of features; and
- can express changes in execution behaviors and needs using attributes.

The intent of our novel framework for building such object-based abstractions is to:

- explore performance issues;
- offer mechanisms for achieving configuration that are lightweight and of general applicability;
- pursue flexibility simultaneously at many levels (ranging from user level objects to operating system services) in complex distributed applications;
- separate basic functionality from configuration issues, both being encapsulated in different components of the framework; and
- promote a model for designing flexible systems and reasoning about configuration possibilities.

The framework, COBS$^{\mathsf{OM}}$, has three kinds of elements: (1)*objects*, (2) *configuration entities* and (3) *configuration channels*, which integrate (1) and (2) during execution. An *object* is described by an IDL interface[25] and an implementation module providing code for its methods. A *configuration entity* encapsulates the information needed to carry out actions related to configuring a given characteristic of an object. It is built separately from the object; the idea is that in the same way that we want to have classes of objects available when building applications, we also want to structure our flexible systems in a manner that classes of configuration may be reused. The application designer composes a configurable/flexible application element by coupling basic functionality (*objects*) to the components that describe each configuration aspect being explored (*configuration objects*). This approach makes "configuration" a first class element in our programming model. The usual object-oriented programming model, that comprises a collection of objects that communicate through method invocations, is now extended to include the presence of *configuration objects* that, once associated with an object, are able to direct the changes in its behavior. The association between *object* and *configuration object* via *configuration channels* is explicitly and dynamically specified. The configuration channel provides information that determines how the interaction between the objects and configuration objects is implemented.

Before coming up with detailed mechanisms for the interaction (*i.e.,* how to make objects and configuration entities work together), we address a more elementary problem: *when a designer finds a convenient configuration entity in the configuration library, how can she assure that it can be integrated with the particular object to be enhanced with flexible behavior?* A complete answer to this question would imply the use of knowledge about the component's semantics. In general, applications where high performance is important do not employ programming environments, languages or tools that make information about the software elements at such a level of detail. Research results in analyzing class hierarchies for reuse[3] indicate that, even with appropriate formal models and specification levels, the solution can be too computationally intensive to be used at runtime. In COBS$^{OM}$, we adopt a simple solution for checking if objects and configuration entities can be integrated into a configurable software element. Namely, we define *compatibility* in terms of the the basic object's interface and the information available in the configuration entity's description. The configuration entity specifies its requirements on the object by enumerating the methods it expects to have available in the object's interface. We refer to these methods as *required methods*; they represent hooks that can be used by the configuration object in order to (1) get information from the object and (2) impose behavior or state changes that may be needed so that configuration actions can be carried out.

Figure 2 pictures three basic objects (*PriorityQueue, SimpleDataBase,* and *LinkedList* and one configuration entity enumerating its required methods. Objects *PriorityQueue* and *SimpleDataBase* are both compatible with the (partially depicted) configuration entity; object *LinkedList* is not.

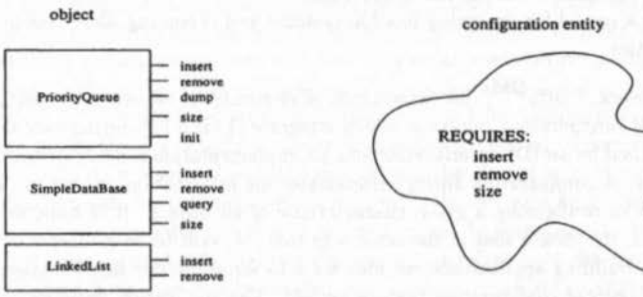

Figura 2: Exemplifying the compatibility notion between object interfaces and configuration entities

Notice that the design and implementation of a configuration entity module should not rely on any specific information of the object being configured, since at compile time, the only information about such an object is that it provides the *required methods*.

Multiple objects from different classes may be simultaneously attached to a given configuration entity, thereby allowing a single configuration object to manage the configuration of multiple basic objects. When the configuration object invokes one of the *required methods*, the runtime system has to map this invocation into the respective method belonging to the specific object that is interacting with the configuration object when the invocation is issued.

*Configuration entities are objects*, therefore they also offer an interface. The methods in this interface represent configuration actions that can be initiated by explicit application demand. In this sense, the configuration entity is expanding the basic object's interface by offering configuration-specific methods.

The configurable objects composed by the association of objects and configuration objects can be varied at runtime, with parts being *efficiently* added or eliminated dynamically. More importantly, this association can be specified at the operation level, allowing a single object to carry out very different configuration approaches, accordingly to which method is being invoked.

*Configuration channels* abstract how invocations on the object interact with the configuration entity. They represent the link integrating objects and configuration entities, and they define how implicit configuration actions are activated during execution. Configuration channels are defined in the specification of the configuration entity, *i.e.*, for each available channel to a configuration entity therfollowing information is provided: (1) its identification, (2) the code to run once the channel is activated, and (3) characteristics determining how the channel abstraction is realized in an implementation.

Figure 3 portrays one object, three configuration entities, and configuration channels. This example shows how configuration entities and configuration channels can be used to compose a flexible distributed, persistent, monitored queue. In our previous work we showed that employing such a configurable queue in the implementation of a branch and bound algorithm for the Traveling Salesperson Problem can result in significant performance gains[26]. The queue behavior can be dynamically changed by removing and inserting configuration channels, and by changing the values of attributes that specify how a configuration entity acts. For example, by eliminating the link between one operation in the queue and the *Monitoring* configuration entity, selective monitoring can be achieved. By varying attributes values of the *FragmentedList* object, we can change how queue distribution is carried out.

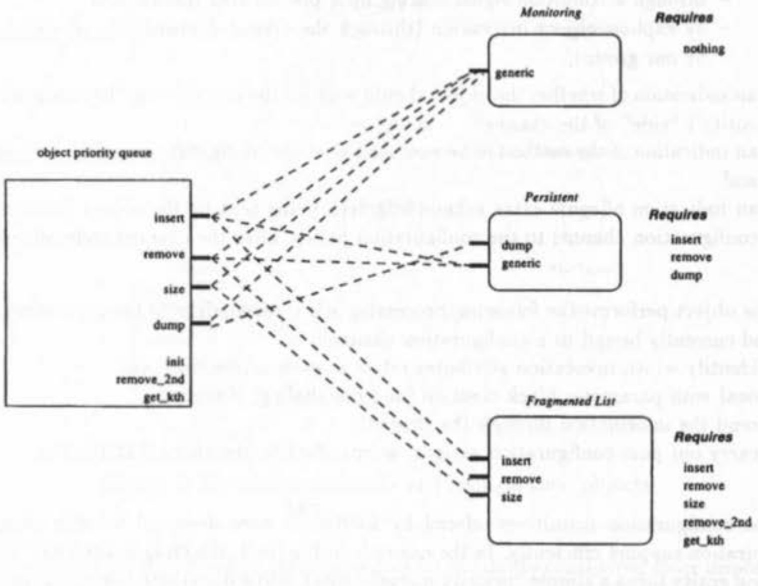

Figura 3: Configuration channels associate objects to configuration entities

When a configuration object is associated with an object (in our interface, via the *binding* call), it becomes possible for the configuration object interfere with the behavior of the basic object. Channels abstract association between entry points in the object (methods) and configuration (channel specification in the configuration entity).

Channels can be opened in two ways:

- automatically, when a configuration entity is bound to an object, either at compile time or during execution. This may be useful for composing applications where few configuration channels are available, and there is an "all-to-all" relationship among objects and channels (Figure 3); and
- by explicit instruction from the application, which specify which method should be associated to which channel.

Parameters that determine channel's implementation behavior include: (1) attribute values in the configuration entity description, (2) values provided by the object when it initiates an *open_channel* operation, and (3) the runtime library default values. All such attribute values can be overridden at any time, thereby changing the channel's behavior. The channel attributes available in COBS$^{OM}$ deal with the following issues:

- reuse of parameter blocks;
- how to send information (operation parameters, attributes) through the configuration channel:
  - through a function call;
  - forking a thread for running the channel code;
  - through a condition signal waking up a pre-existing thread; and
  - by explicit object invocation (through the *Object Transport Layer* developed by our group);
- an indication of whether the object should wait for the execution in the configuration entity's "side" of the channel;
- an indication of the method to be executed after the configuration entity's actuation; and
- an indication of some extra acknowledgment being sent by the object through the configuration channel to the configuration policy, after the channel code returns.

The object performs the following processing when responding to the invocation of a method currently bound to a configuration channel:

- identify which invocation attributes relate to each active channel;
- deal with parameter block creation (and marshaling, if needed);
- send the information through the channel;
- carry out post-configuration actions, as specified by the channel attributes.

The configuration primitives offered by COBS$^{OM}$ were designed to offer complex configuration support efficiently. In the example in Figure 3, the "fragmented list" configuration entity turns a simple "priority queue" object into a distributed one: it generates name server information to be given to the transport layer, it creates the "priority queue" fragments on the multiple nodes, it allocates one local configuration entity to each of the nodes, and it manages distribution transparently to the queue object's user. COBS$^{OM}$'s support facilitates the development of such a configurable abstraction, but the composed element would not be useful if the overheads imposed by configuration interactions were high.

Configuration entities are implemented through objects, and therefore they can also be configured by association with other configuration entities, resulting in complex hierarchies of objects and configuration objects. Figure 4 illustrates a hierarchy composed by three kinds of basic objects: (1) *BTree-A* and *BTree-B* are instances of a binary tree data structure, (2) *Database* is a complex object that has a collection of *BTrees* as part of its state, and (3) *Device-1* and *Device-2* represent output devices. The other objects in the figure, represented by ellipses, are configuration objects. *PersistenceConfig* is adding persistence to the behavior of *BTree-A*, *BTree-B*, and one of the *Database*'s internal binary trees. *PersistenceConfig* itself is linked via channels to configuration objects *CheckPoint* and *IncrementalLog*. As a result, the way in that persistence actions are carried out can be changed dynamically. *Device-1* is a component of *PersistenceConfig* that can be associated with *SocketOutput* or *FileOutput*. These two configuration objects manage multiple *Device* instances by storing one file descriptor (*fd*) for each basic object they configure.

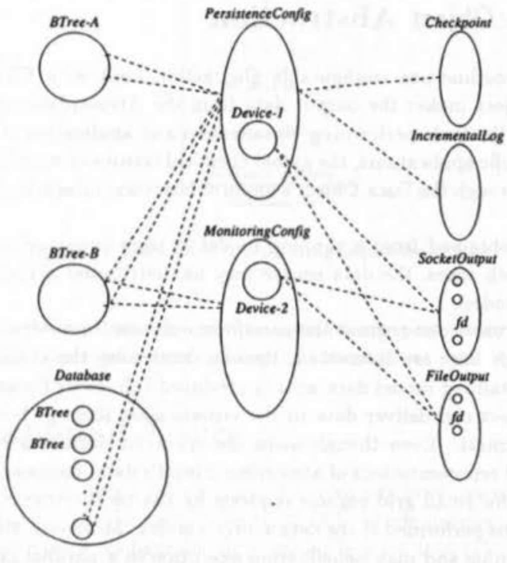

Figura 4: Hierarchy of objects and configuration objects

In summary, $\mathsf{COBS}^{\mathsf{OM}}$ offers the approaches for flexibility considered more important in the literature:

- *Parametric variation*: the application ensures that relevant information about current demands and preferences about an available service are transferred to the service, which will then take appropriate actions to meet their requirements. This appears in the COBS$^{OM}$ through the use of *attributes* that are passed from invocations to the configuration objects. Each configuration object defines and enforces semantics to its attribute values.

- *Interposition*: the flexible service maintains a fixed interface, but application-level code can be interposed between the uses of the service and the available service implementation. In this fashion, the application developer incrementally changes the semantics of a service, without altering the service itself. In COBS$^{OM}$, this can be easily achieved by dynamic association of configuration objects with specific methods (services) via configuration channels that are created using the default values for channel attributes.
- *Synthesis*: the application developer is allowed to synthesize an additional service, by specifying both its interface and its implementation. These new services should be treated as basic services, for example as an extension to the operating system or basic parallel programming support. An example of this is the synthesis of a distributed persistent monitored queue at runtime.

## 4 The Data_Object Abstraction

In this section we outline one configurable abstraction built with COBS$^{OM}$: the Data_Object. This object makes the output data from the Atmospheric Application (Section 1) available to the tools performing visualization and application steering[8].

Like most scientific applications, the global chemical transport model produces a large amount of data. Through the Data_Object's uniform interface, interactive tools can access this data flexibly:

- Data can be obtained from a running model or from previous executions' stored results. In both cases, the data source may be distributed across several computing/storing nodes.
- Data_Object's users can request the specific simulation time steps and atmospheric levels in which they are interested, thereby decreasing the communication costs involved in attaining model data as it is produced (all levels for each time step).
- The Data_Object can deliver data to the visualization tools in both spectrum and grid point formats. Even though using the spectrum format generally results in more compact representations of atmospheric level's data, communication costs can be decreased for small grid regions requests by having the spectrum-to-grid point transformations performed at the data source's nodes. Moreover, the transformation is time consuming and may benefit from execution in a parallel platform.
- multiple users can simultaneously examine the data.

Figure 5 portrays the Data_Object layout when data is retrieved from two files produced by the model. The object is fragmented such that the interface to the application code (*e.g.,* visualization tool) is available in fragment *A*. When a request for grid point data is issued, *A* will infer which node(s) store the data and then invoke the appropriate(s) object fragment(s). The application can dynamically attach filters to object fragments, thereby refining the data before it is sent through the communication link. In Figure 6, data is retrieved from a running model through sockets. The Data_Object initiates its execution by opening connections to the sockets where the data model is depositing output data as a stream of events. Each event contains spectrum information about all levels for a given simulation time step. Data_Object is fragmented in two kinds of objects: interface objects and *concentrators*. The interface objects ($X$ and $Y$ in Figure 6) are equivalent to the object $A$ (Figure 5), each one serving one Data_Object's user. Object fragments $Z$ and $T$ act like *concentrators*, meaning that they temporarily store and manage the data being produced by the model. The number and locality of concentrators can be dynamically configured. They can serve multiple interface objects.

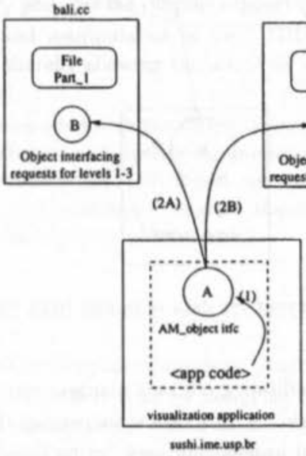
bali.cc
File Part_1
B
Object interfacing requests for levels 1-3
aruba.cc
File Part_2
C
Object interfacing requests for levels 4-6
(2A)
(2B)
A
(1)
AM_object itfc
<app code>
visualization application
sushi.ime.usp.br

Figura 5: Data_Object obtaining input from files

Experiments are being made in order to measure the performance impact of using Data_Objects, both in terms of access' latency and throughput rate.

## 5 Implementation Issues

COBS$^{OM}$'s design incorporated insights derived from our previous work on supporting configuration in multiprocessor environments[20, 26]. While developing CTK (a toolkit for building multiprocessor configurable objects) we learned that (1) complex configurable abstractions require *n-to-n* relationships between objects and configuration components in order to achieve efficient use of resources and (2) support for object replication facilitates exploration of locality. The CTK toolkit provides compiler support for a C-extension programming language that combines object implementation and configuration components description into a single module. Besides controlling initialization and usage of the underlying thread system, the CTK's runtime system entails detailed information about classes, objects, operation's signatures, and compile-time configuration setup. In COBS$^{OM}$, the amount of static information about objects manipulated during runtime

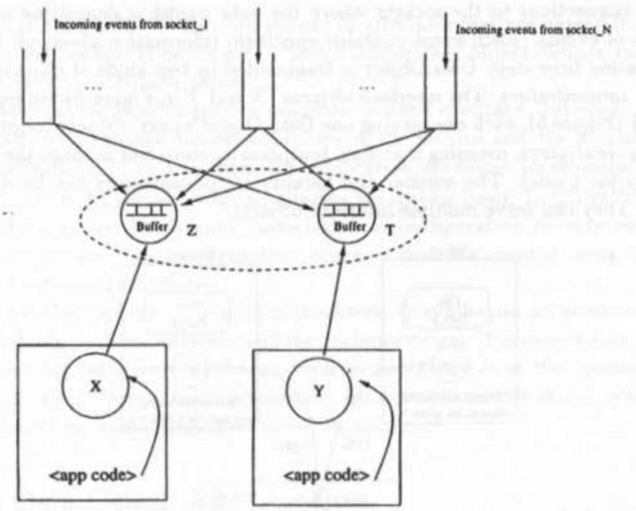

Figura 6: Data_Object layout for data obtained from running model

is kept to a minimum, thereby facilitating efficient management of distributed objects. COBS$^{OM}$ takes a library approach, by this means increasing the model's strength in three ways: (1) legacy objects and implementations can be integrated with COBS$^{OM}$'s objects, (2) configuration appears as a first class element in the model, *i.e.*, , application's programmers are able to manipulate configuration abstractions and freely alter their composition, and (3) parts of COBS$^{OM}$'s functionality (*e.g.*, the support for object fragmentation) can be easily reused.

Objects are described by an IDL interface, and the user provides an implementation model which associates code with the methods offered in the interface. Using an IDL compiler front end, we have constructed an IDL to C compiler which generates: (1) a header file describing defined types and method prototypes, and (2) a meta-description of the interface. This intermediate description form is created so that the IDL to C compiler front end is decoupled from our object model support. The usual IDL attribute semantics is offered and mapped to the attribute implementation offered by Eisenhauer's work on the COBS project[2].

COBS$^{OM}$ offers some tools to facilitate the developer's job. For example, the tool code_gen consumes the meta-description of an IDL interface and generates class specific implementation routines for the creation of objects, allocation of parameter blocks for the methods, and invocation of methods. It also exports the interface description to the *Interface Repository*, so that available information about the class can be queried at runtime for checking composition compatibility.

The configuration entity descriptions are processed by the tool conf_gen. As is the case with *code_gen*, code is generated from the description to deal with the manipulation of configuration entities, and a configuration description file is stored in the repository.

In order to facilitate the process of building applications, we allow their specification in terms of which classes (of objects and configuration objects) it uses. From this description, the tool app_gen generates a makefile, ensuring that all the necessary compilation steps and COBS$^{OM}$ tools are applied. Another relevant function of *app_gen* is to generate code for the initialization step in building a distributed application; this is accomplished by building a specific *object server* for the application. This server is the available interface for requesting the creation of objects on a remote node.

The framework library provides the runtime support for objects, configuration objects, configuration channels, and manipulation of basic IDL types. COBS$^{OM}$ can be used with a threads package, thereby allowing the use of its concurrency/parallelism support through the object model.

COBS$^{OM}$ has been integrated with the *Object Transport Layers* (COBS$^{OTL}$) package that offers support for efficient and flexible remote invocation, so objects can be distributed in a network of workstations. With just one call and the specification of a few attributes, it is possible to transform a program that utilizes only local objects into a program where the objects reside on different notes.

## 6 Conclusion

The framework presented in this paper provides a programming model and environment where flexible software can be developed by designing configurable objects. COBS$^{OM}$'s efficient runtime support enables performance improvements by configuration of program components during their execution. Program configuration is attained without compromising the encapsulation or the reuse of software abstractions, by explicitly separating the type-dependent object functionality from its properties subject to configuration, including its performance, reliability, and timing properties. *Objects* and *configuration objects* can be combined in complex ways, and the composition can be changed dynamically without the imposition of unreasonable overheads. Our experience in building configurable objects, in particular the Data_Object presented in this paper, indicates that COBS$^{OM}$ is suitable for building high performance distributed and parallel objects that can achieve flexible and complex behavior in runtime. Moreover, the programming model offered by COBS$^{OM}$ encourages incremental design and reuse of components.

## Acknowledgments

Greg Eisenhauer has contributed a lot to this work with valuable ideas and full prompt support in the use of OTL/pbIO libraries. This research has been supported in part by CNPq-Brazil (No 200199/90-1), DARPA (DABT63-95-C-0125), and FAPESP (523037/96-1).

## Referências

[1] B. Agnew, C. Hofmeister, and J. Purtilo. Planning for change: A reconfiguration language for distributed systems. In *Proc. of the Second International Workshop in Configurable Distributed Systems*, pages 15–22. IEEE Computer Society Press, May 1994.

[2] M. Ahamad and K. Schwan. The COBS Project. http://www.cc.gatech.edu/systems/projects/COBS, 1995.

[3] H. Astudillo. *Evaluation and Realization of Modeling Alternatives: Supporting Derivation and Enhancement.* PhD thesis, College of Computing, Georgia Institute of Technology, April 1996.

[4] R. Campbell, V. Russo, and G. Johnson. Choices (class hierarchical open interface for custom embedded systems). *ACM Operating Systems Review*, 21(3):9–17, July 1987.

[5] *Proceedings of the 3rd International Conference on Configurable Distributed Systems.* IEEE Computer Society Press, May 1996.

[6] H. Custer. *Inside Windows NT.* Microsoft Press, Redmond, Washington, 1993.

[7] S. Danforth and I. Forman. Reflections on metaclass programming in SOM. In *Proc. of OOPSLA'94*, pages 440–452. ACM Press, October 1994.

[8] G. Eisenhauer, B. Schroeder, and K. Schwan. From interactive high performance programs to distributed laboratories: A research agenda. In *Proc. of the SPDP'96 Workshop on Program Visualization and Instrumentation*, October 1996.

[9] G. Eisenhauer and K. Schwan. Parallelization of a molecular dynamics code. *Journal of Parallel and Distributed Computing (SPDT)*, 34(2), May 1996.

[10] D. R. Engler, M. F. Kaashoek, and J. O. Jr. Exokernel: An operating system architecture for application-level resource management. In *Proc. of the 15th Symposium on Operating Systems Principles.* ACM Press, December 1995.

[11] G. K. et al. Open implementations: A metaobject protocol approach. In *Proc. of the 9th Conference on Object-Oriented Programming Systems, Language, and Applications*, 1994. Tutorial notes.

[12] I. Froman, S. Danforth, and H. Madduri. Composition of before/after metaclasses in SOM. In *Proc. of OOPSLA'94*, pages 427–439. ACM Press, October 1994.

[13] G. Hamilton, M. Powell, and J. Mitchell. Subcontract: A flexible base for distributed computing. In *Proc. of the 14th ACM Symposium on Operating Systems Principles*, pages 69–79. ACM Press, December 1993.

[14] G. Kiczales and J. Lamping. Operating systems: Why object oriented? In *International Workshop in Object Oriented Operating Systems*, pages 25–30, December 1993.

[15] T. Kindberg, A. Sahiner, and Y. Paker. Adaptive parallelism under Equus. In *Proc. of the 2nd International Workshop in Configurable Distributed Systems*, pages 172–182. IEEE Computer Society Press, May 1994.

[16] T. Kindler, K. Schwan, D. M. Silva, M. Trauner, and F. Alyea. Parallelization of spectral models for atmospheric transport processes. *Concurrency: Practice and Experience*, 8(9):639–666, November 1996.

[17] P. Maes. Concepts and experiments in computational reflection. In *Proc. of OOPSLA '87*, pages 147–155. ACM Press, October 1987.

[18] J. Magee, N. Dulay, and J. Kramer. A constructive development environment for parallel and distributed programs. In *Proc. of the Second International Workshop in Configurable Distributed Systems*, pages 4–14. IEEE Computer Society Press, May 1994.

[19] H. Massalin. *Synthesis: An Efficient Implementation of Operational System Services*. PhD thesis, Columbia University, 1992.

[20] B. Mukherjee, D. Silva, K. Schwan, and A. Gheith. Ktk: kernel support for configurable objects and invocations. *Distributed Systems Engineering Journal*, 1:259–270, 1994.

[21] A. Paepcke, editor. *Object-Oriented Programming – The CLOS Perspective*. MIT Press, 1993.

[22] R. Rashid, D. Julin, and et al. Mach: A system software kernel. In *Proc. of the 34th IEEE Computer society International Conference (COMPCON 89)*, pages 176–178, February 1989.

[23] D. Schmidt. The adaptive communication environment. In *Proc. of the 11th Sun User Group Conference*, 1993.

[24] K. Schwan, A. Gheith, and H. Zhou. Chaos-arc: A kernel for predictable programs in dynamic real-time systems. In *7th IEEE Workshop on Real-Time Operating Systems and Software, Univ. of Virginia, Charlottesville*, pages 11–19, May 1990.

[25] J. Siegel. *CORBA Fundamentals and Programming*. John Wiley & Sons, Inc., 1996.

[26] D. M. Silva and K. Schwan. CTK: configurable object abstractions for multiprocessors. Technical Report GIT-CC-97-03, Georgia Institute of Technology, Atlanta, GA 30332, January 1997. Submitted to IEEE Transactions on Software Engineering.

[27] I. Sommervile. *Software Engineering*. Addison-Wesley Pub Co, 4th edition, 1992.

[28] Y. Yokote. The Apertos reflective operating system: The concept and its implementation. In *Proc. of OOPSLA '92*, pages 414–434. ACM Press, October 1992.