Configurable Distributed Retrieval of Scientific Data

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

Karsten Schwan Greg Eisenhauer
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{schwan,eisen}@cc.gatech.edu

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. 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 runtime execution adjustments of the basic mechanisms that influence them. We introduce the basic elements of the model. We also present Data_Object, a complex configurable object that provides flexible access to data output from a high performance parallel and distributed scientific application.

1. Introduction

New categories of computing applications like multi-media systems, the wide area distribution of information across the Internet, groupware, and mobile computing have 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 execution of application behaviors specialized for these environments.

Our work has been addressing *runtime flexibility* as a crucial requirement for current and emerging application domains, by incorporating adaptation capabilities into software components. Examples range from the adaptation of communication protocols to changes in network or application behavior [12], to the runtime alteration of operating system abstractions [15], to application-level changes performed by adaptation heuristics [20, 9] or by end users themselves [4]. At the application-level, our research offers the notion of *configurable objects* as a means of achieving flexible software in which runtime execution adjustments lead to improved performance.

The work described in this paper explores configurability issues in distributed environments for a specific problem: the high performance retrieval of data produced by a large scale scientific application. The goal is to integrate at runtime the output generated by a high performance parallel scientific code with its clients (e.g., data visualizations or collaborative frameworks), resulting in the increased availability of data and in reduced network communication costs. To attain this goal, we have developed a system for the construction and manipulation of dynamically configurable objects, called COBS-OM (COBS [1] project Object Model). This system offers programming environment and runtime support for object configuration. It addresses performance issues by providing novel abstractions for incorporating runtime flexibility into a distributed object program in a methodical fashion.

The novel contributions of COBS-OM are its offering of *configuration objects* and their association with targets via *configuration channels*. With these abstractions and by using the adaptation techniques of code interposition, object fragmentation, and parametric variation, we are able to increase flexibility and realize performance improvements in the execution of the aforementioned high performance data generation/access application.

Section 2 describes the specific configuration requirements of the data access problem we have investigated. Section 3 states the COBS-OM model of configurable objects and describes its design and implementation. The model's application to the data object results in the configurable data object described in Section 4. Conclusions and related work are elaborated in Section 5.

2. Flexible Access to Distributed Data

In collaboration with atmospheric scientists at Georgia Tech, we have developed a parallel and distributed global chemical transport model [11] (GCTM) capable of running in heterogeneous high performance computing environments (Figure 1 depicts the environment available in our laboratory). Models like this are important tools for answering scientific questions concerning the distribution of chemical species such as chlrofluorocarbons, hydrochlorofluorocarbon, and ozone. When investigating these questions, scientists wish to extract from this model, at runtime and subsequent to each simulation time step, data detailing current winds, wind directions, and chemical species concentrations at each of up to 37 levels of atmosphere (a level typically corresponds to some range of barometric pressure). Toward this end, they employ various visual displays, including simple 2D slices at specific barometric levels of the atmosphere or complex 3D visualizations of specific atmospheric volumes.

The software technologies presented in this paper address the tasks of efficiently retrieving, delivering, and transforming this potentially large scale and naturally distributed data. Specifically, using the COBS-OM framework for developing configurable objects, we have developed a configurable object that offers flexible and efficient access to the model's data. In this case, flexibility is desired for two reasons: (1) to deal easily with different configurations of the parallel model in terms of degrees of parallelism and speeds of the machines to which its components are mapped, and (2) to react to changes in data access and manipulation by the end users viewing and manipulating such data via the visualization environments they employ.

Through the Data_Object's uniform interface, interactive tools can access GCTM data flexibly:

- Data can be obtained from a running model or from stored results of previous executions. In either case, 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 data at the granularity produced by the model (all levels for each time step).
- The Data_Object can deliver data to the client tools in both spectral and grid domains. Although the spectral format is a compact representation for atmospheric level's data, in the case of small grid regions, communication costs can be decreased by having the spectral-to-grid point transformations performed at the data source's nodes. Moreover, this transformation is time-consuming and benefits from its execution on a parallel platform [11].
- Multiple users can simultaneously examine the data, by simply invoking the Data_Object's interface concurrently. This demonstrates Data_Object's utility for implementation of the diverse collaboration roles sought by multiple end users interacting with each other and the model via diverse user interfaces.

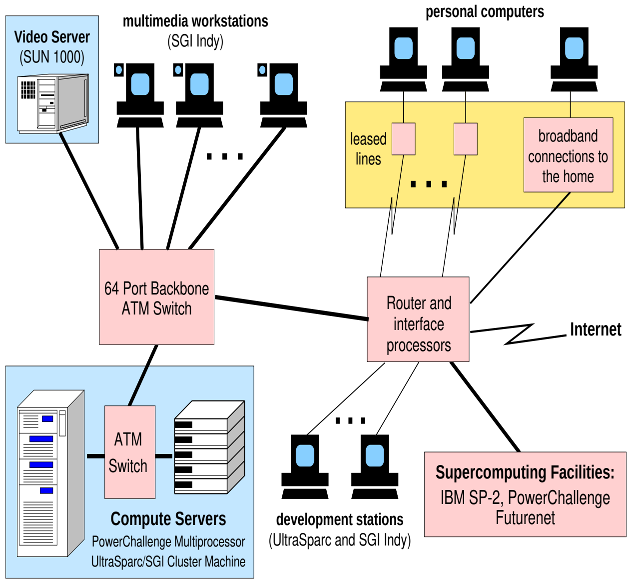

Figure 1. The computing environment in the Distributed Laboratories Project at Georgia Tech.

As with other recent efforts to organize and access large scale parallel and distributed data [5], Data_Object supports simple and application-specific queries on distributed data. Namely, it offers a natural interface for requesting specific sets of data. For example, for the interactive steering support tool developed by Heiner and Zou [3], the Data_Object method "get_grid (*latitude/longitude range*, *range of levels*, *range of time steps*)" is sufficiently general to address all the existing tool requests for application data. Moreover, the level of abstraction in the method signature matches the tool's view of data.

Data_Object's functionality is not limited to that offered by parallel I/O systems. Instead, by using the techniques of interposition, parametric variation, and object fragmentation, Data_Object can efficiently offer the specific functionality needed by its diverse clients and match the heterogeneous and distributed nature of its underlying computing environment:

1. *Code interposition* – Data_Object permits the associ-

ation of application-specific computations with such data accesses, thereby enabling the implementation of *active input/output streams*, as explored further in our current work. Application-dependent conversion may be enhanced with additional conversion to deal with the heterogeneity of machines' basic data types across data generators vs. users. Factoring this conversion out of the accessing tool increases modularity and facilitates reuse. Last and perhaps most importantly, the "data examiner" client is able to specify new filters and associate them with Data_Object dynamically, in the spectral or grid domains. This enables semantically meaningful reductions in the amounts of data accessed and transferred and the sizes of computations performed on such data.

2. *Parametric variation* – by varying the semantics of its invocations, Data_Object can be extended to support application-specific implementations of collaboration across multiple visualizations using it.

3. *Changes in object representation* – Data_Object may be internally fragmented (in terms of its state and computations) across the multiple machines on which data is generated or stored, thereby enabling efficient access to and transformation of distributed data. Specifically, since the object's internal fragmentation is flexible and transparent to the object's users, exploration of techniques for minimizing communication and access costs (*e.g.,* data caching) can be carried out independently of the tool's development.

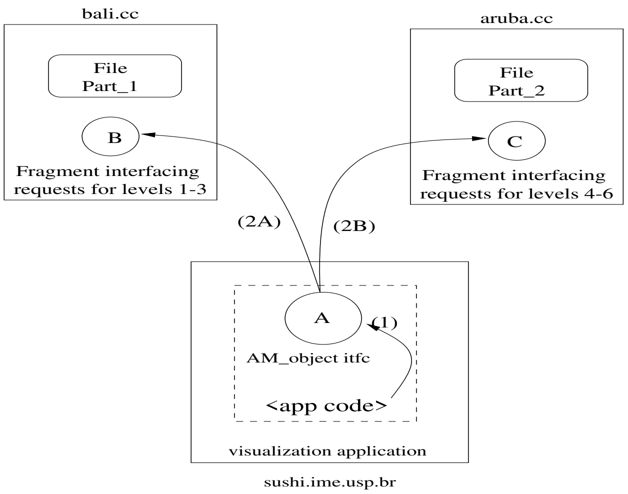

Figure 2. Data_Object with input from files

Figures 2 and 3 present a high level picture of the Data_Object as a composition of multiple object fragments. These figures will be refined in Section 4, where the Data_Object's design is described in more detail.

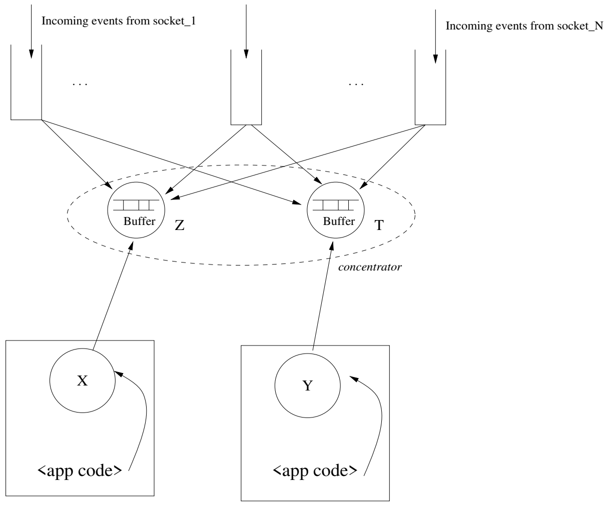

Figure 3. Data_Object with input from running model

An important characteristic of COBS-OM is its support of dynamic vs. static configuration. Namely, objects like Data_Object need not be configured once, at their time of instantiation. Instead, such configuration may occur throughout each object's existence, by using COBS-OM's configuration techniques. To illustrate, consider Figure 2, which portrays the Data_Object initial 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 – *code interposition* – to object fragments, thereby refining the data before it is sent through the communication link. In Figure 3, data is retrieved from a running model through sockets. The Data_Object initiates its execution by opening connections to the sockets through which the data model is depositing output data as a stream of events. Each event contains spectral information about a set of levels for a given simulation time step. Two kinds of components are pictured: interface objects and *concentrators*. The interface objects (*X* and *Y* in Figure 3) are equivalent to the object *A* (Figure 2), 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, and they can serve multiple interface objects. Parametric variation may also be employed for runtime adaptation of the Data_Object: the client can dynamically switch between a configuration suited for accessing recent data (via sockets, as in Figure 3) and a configuration that retrieves past data from files (as in Figure 2).

3. The Programming Model

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. For example, in the Data_Object configuration picture in Figure 3 there are objects responsible for the offered basic functionality (X and Y) and objects related to varying implementation (the "concentrators") in terms of distribution and use of resources such as memory and networking connections.

COBS-OM allows the construction of object abstractions that: (1) encapsulate some basic functionality; (2) are able to accommodate dynamic changes in how their functionality is implemented; (3) permit the dynamic addition or subtraction of features; and (4) can express changes in execution behaviors and needs using attributes.

All these characteristics have been useful during the incremental design of the object for flexible access of simulation data presented in this paper (the Data_Object). The intent of our novel framework for building such object-based abstractions has been to:

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

The first two issues address the need for deriving concrete results from the construction of configurable applications; the others relate to facilitating the development of configurable software.

The framework, COBS-OM, has three kinds of elements: (1) *objects*, (2) *configuration entities* and (3) *configuration channels*, which integrate (1) and (2) during execution. An *object* comprises the basig funcionality being offered. 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 configuration classes 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.

A basic composition problem is to determine which configuration objects can be integrated to which basic objects, and what are the integration effects. A complete answer to this problem requires the use of knowledge about each component's semantics. In general, programming environments, languages or tools for high performance applications do not make information in such a level of detail efficiently available 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 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. For each pair <basic object class, conf. object class>, compatibility is checked at most once, no matter how many configuration channels connecting the pairs are created or destroyed. 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 code invokes one of the *required methods*, the runtime system has to direct this invocation to the specific object acting as the current subject of configuration.

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 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 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.

Discussion COBS-OM novel aspects rely on the flexible association between basic objects and configuration objects. The latter, like meta-objects [10], can change how the invocations to the basic object are carried out. The application programmer can directly request n to m associations or configuration object actions; these kinds of interactions are not possible with meta-objects. Spring subcontracts [7] can also alter object behavior by changing basic object mechanisms. They differ from our model in many ways:

- subcontracts are used by the object model, not by the basic object implementor. For the programmer, all that is available is the choice of which subcontract to use with a type. This specifies a “flavor” for the object, since it defines how the object support mechanisms will run. Once an object is created, application programmers can not change the subcontract associated with it;
- only one subcontract can be associated with a given object; and
- the interaction pattern between type and subcontract is fixed. For example, the application can not make the subcontract to skip “invoke”. Only the subcontract itself can control its methods.

COBS-OM Implementation: runtime architecture
Figure 4 pictures the interaction between application code and COBS-OM runtime support library. The solid arrows represent request of services, with result information returning to the requester; dashed lines pictures data output.

The framework relies on a repository of component descriptions (updated at compile-time) and data structures describing object compositions. The first prototype included ad hoc marshaling for IDL types; the current implementation uses PBIO [2] support for transport between different address spaces.

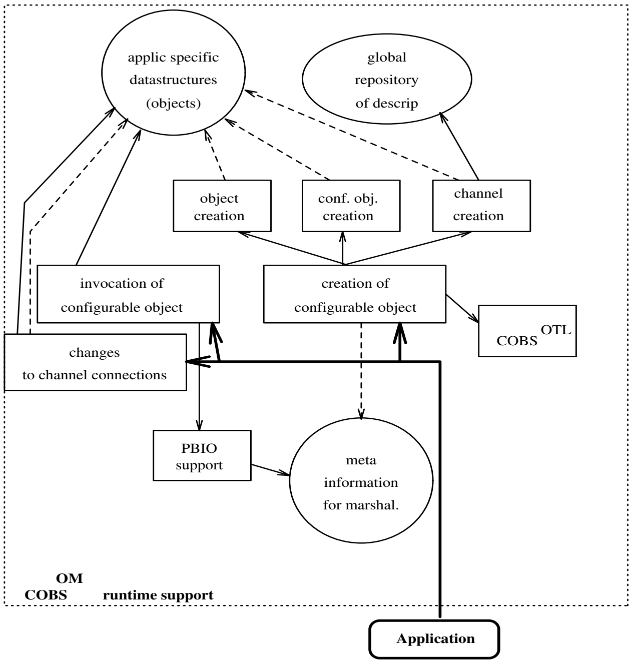

Figure 4. COBS-OM runtime architecture

The framework library provides 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 mechanisms in the development of COBS-OM objects. The current implementation has been tested with Cthreads [15] and Solaris 5.5 Pthreads.

Another relevant function carried out by the framework is the activation of initialization steps when distributed applications are being built up. The current implementation has been integrated with the *Object Transport Layer* (COBS-OTL) package, which offers support for efficient and flexible remote invocation, so that objects can be distributed across networked nodes. By adding an extra call and specifying a few attributes, applications designed and tested with only local objects become distributed. The interfaces to these objects remain the same, regardless of the nodes on which the COBS-OM objects reside. The implementation relies on COBS-OTL for management of object naming, efficient identification of local object references, and propagation of server information (host name, port number) among the nodes participating in the application.

COBS-OM tools Object interfaces are specified using the *Interface Description Language* (IDL) [18]. The object developer provides an implementation module which associates code with the methods present in the interface. Using an IDL compiler front end available from the OMG site [16], we have constructed an IDL to C compiler.

The tool code_gen consumes the description of an IDL interface and generates class-specific implementation routines for the creation of objects, parameter block allocation, and method invocation. For each IDL complex type (sequence, array, or structure) present in the interface, code_gen generates auxiliary routines for the marshaling/unmarshaling tasks. It also extends the usual IDL attribute semantics so that attribute values can be uniformly retrieved from and propagated to lower application levels and subsystems. Moreover, code_gen exports the interface description to the *Interface Repository*, so that available class information can be queried at runtime and composition compatibility can be checked. The PBIO library [2] is again used to achieve inter-operability among descriptions generated by heterogeneous hardware. The tool code_gen generates dispatchers, object creation attributes, and remote invocation information.

The configuration entity descriptions are processed by the tool conf_gen. As with code_gen, code is generated from the description to deal with the manipulation of configuration entities. Also, a configuration description meta-information file is produced and 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) they use. From this description, the tool app_gen generates a makefile, ensuring that all necessary compilation steps and COBS-OM tools are applied.

Support for Fragmented Objects The Object_Server is an object built with COBS-OM that can be invoked from any application. It implements a database for keeping information about COBS-OM objects. By querying the Object_Server, applications can retrieve name information about the objects they intend to use. Global, application-specific, and host-specific searches are available. The Object_Server is a multi-threaded object. Its methods can be executed synchronously or asynchronously.

The Object_Server functionality includes basic support for fragmented objects. In COBS-OM, it is possible to create complex objects whose components are distributed across multiple nodes. Invocations of a fragmented object can be automatically delegated to a local fragment, if present. Applications can register fragments in the database and retrieve information about fragment distribution. The Object_Server interface includes a method for inserting fragments into the database that will block the invocation until all participating fragments are available. This facilitates the developer's task by providing implicit synchronization of composite object creation.

4. The Data_Object

This section provides details about the Data_Object, a configurable abstraction that allows for flexible integration of the Atmospheric Application output data to visualization, steering, and collaboration tools.

Design Data_Object is a complex object composed of the following components: a *data_retriever*, a *transformer*, an *application_interface*, and a (potentially empty) set of *data_receiver* objects. Each one of these components is a configurable object implemented via objects, configuration objects, and configuration channels and they relate to different configuration facets: distribution, interposition, and caching. The components may be distributed across computing, storage, and visualization[1] nodes. This distribution can be altered at runtime. Some of these components are fragmented themselves, increasing the opportunities for configuring object distribution. In addition to these object parts, *filters* may be dynamically created and attached to any of the components, so that the client can use interposition code to adapt the retrieved data to its own needs.

The *data_retriever* abstracts the data sources. Usually it is configured at initialization time to access either files or sockets. The data is distributed in terms of atmospheric levels and/or time steps. The distribution can change at runtime (*e.g.,* more time steps are made available in new files).

The *transformer* object is responsible for performing the spectral to grid computation. It receives data from the *data_retriever* fragments and returns the associated grid points to the *application_interface* object.

The *application_interface* receives requests from the tool(s). It coordinates the components' behavior in order to retrieve, transform, and filter data.

Filters are configuration entities that may be attached to any of the Data_Object's components via configuration channels. They encapsulate a filtering function that is automatically applied to data arriving through the channel. Filters that support sets of grid point values and generic sequences of numbers as input are currently available.

Finally, the *data_receiver* is another type of object through which a tool is able to obtain application data. It has been designed to provide simple support for propagating the accessed grid point data to a collection of cooperating agents. When a *data_receiver* is created, it is connected to the *application_interface* object, which will forward through the connection all data received. Connections can be temporarily closed by either the *data_receiver* or the *application_interface* object. Using *data_receiver* objects, the teacher-student pattern of collaboration can be achieved.

[1] Visualization tools are just one example of data access applications. The Data_Object has been designed to work with the visualization tool implemented by our group, but it is not restricted to it.

Filters can be attached separately to each *data_receiver*, thereby customizing the "student view".

Figures 5 and 6 depict possible ways in which the Data_Object parts may be distributed. In Figure 5, the data is read from files, the transformation from spectral to grid points is computed within the storing nodes, and no data filtering is performed. The *application_interface* object contains a *distributor* configuration object that forwards data requests to the appropriate *data_retriever* fragment. Figure 6 portrays the scenario in which input data arrives through sockets and it is manipulated by the concentrators, which have attached filters that act upon the data before it is sent to the visualization node. By specifying attribute values, Data_Object's users can transform the object setup described in Figure 5 into the arrangement in Figure 6.

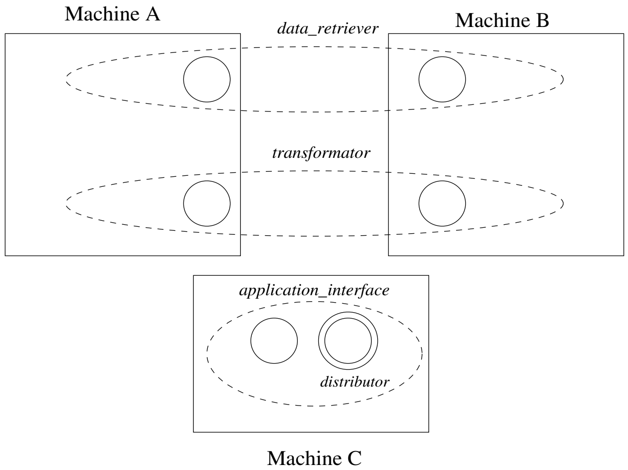

Figure 5. Example of Data_Object configuration

Creation and Adaptation Two approaches for creating a Data_Object are available: (1) via a graphic user interface or (2) directly through the object creation call in the COBS-OM API. In the first approach, the user provides information about the data sources and Data_Object's component distribution using buttons and menus. The values provided by the user are verified (*e.g.,* checks for file existence, socket connections, host names, object redefinition, *etc.*) and interpreted. The system invokes the COBS-OM object creation call, and registers the new Data_Object with the Object_Server. Tools can retrieve a Data_Object *proxy* from the Object_Server, and use it either for requesting data (as in the "get_grid" call) or connecting to a *data_receiver* that passively will receive the data retrieved from the Data_Object.

Data_Object can be configured at runtime by attribute values passed through method invocations. For example, at creation time attributes can be used to specify the type of *data_retriever* to be created, input data characteristics, and distribution of *transformers* across nodes. Whenever Data_Object's methods are invoked, attributes can be passed to the configurable object resulting in adaptations such as *transformer* migration, attachment of new filters to object components, or disconnection of a *data_receiver*.

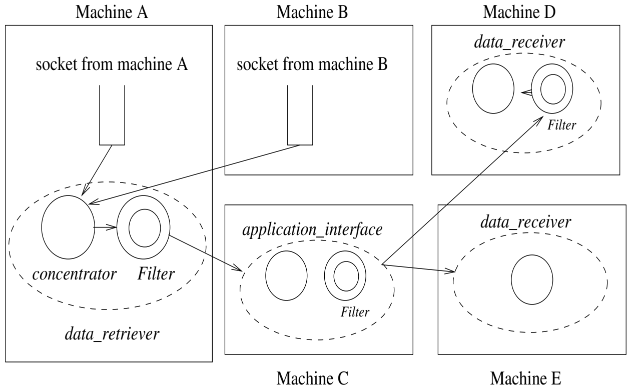

Figure 6. Example of Data_Object configuration

Some of the possible adaptations involve information better expressed through complex data types. For example, while adapting a *concentrator* (configuration entity related to *data_retriever* objects) from a central node to a distributed implementation — in which fragmentation format is derived from level information —, the distribution description can be naturally represented by an arbitrarily sized sequence of structures. The current COBS-OTL attribute implementation does not provide support for complex types. In order to spare the Data_Object's users from the cumbersome task of flattening the complex data structure into a collection of basic attributes, the Data_Object and its components' interfaces have been extended with a set of specific configuration methods (such as expanding the set of nodes storing buferred data). These methods receive configuration descriptions as arguments and change the appropriate attribute values and configuration channels.

5. Related Work and Conclusion

Section 3 briefly discussed the relationship between the configuration programming model being presented and work done on meta-objects (reflective programming) and Spring subcontracts. Other related work includes [17, 21, 6]. The retrieval of data from high performance simulating is related to the current work in parallel I/O. In particular, a declustering technique for maximizing disk parallelism has been extensively studied [14]. The Schooner project [8] defines an interconnection system that can be used to connect visualization tools to a graphics workstation so that data generated by a scientific simulation on a parallel machine can be displayed. Regis/Darwin [13] encourages a component based approach, providing a configuration language. The connections in Regis/Darwin are realized through port objects, which queue messages of a particular type; in our work the connection can vary from function calls to remote method invocation.

Performance Performance experiments have been recently driving some refinements in the design and implementation, so that we can show that the configuration support can be carried out efficiently. The detailed current performance numbers can be found in [19]. Improvements are demonstrated on a cluster of workstations and shared memory parallel machines jointly executing the scientific application and multiple visualizations accessing the data it produces. Microbenchmarks are used to evaluate specific COBS-OM components and their basic performance properties.

Conclusion The framework presented in this paper provides a programming model and environment where flexible software can be developed by designing configurable objects. The basic framework elements, *objects* and *configuration objects* can be combined in complex ways, and the composition can be changed dynamically. 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 This research has been supported in part by CNPq-Brazil (under number 352737/96-4) and DARPA (under contract DABT63-95-C-0125)

References

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

[2] G. Eisenhauer. Portable self-describing binary data streams. Technical Report GIT-CC-94-45, College of Computing, Georgia Institute of Technology, Atlanta, GA, 1994.

[3] 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.

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

[5] R. Ferreira, B. Moon, J. Humphries, A. Sussman, and J. Saltz. The virtual microscope. Technical Report UMIACS CS-TR-3777, University of Maryland, Department of Computer Science, 1997.

[6] 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.

[7] 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.

[8] P. T. Homer and R. D. Schlichting. Configuring scientific applications in a heterogeneous distributed system. *IEE Distirbuted Systems Engineering Journal*, 1996 1996.

[9] R. Jha, M. Muhammad, S. Yalamanchili, K. Schwan, and D. I. Rosu. Adaptive resource allocation for embedded parallel applications. In *Proceedings of the 3rd International Conference on High Performance Computing (HiPC)*, Trivandrum, India, December 1996.

[10] G. Kiczales 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.

[11] 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.

[12] R. Kravets, K. Calvert, P. Krishnan, and K. Schwan. Adaptive variation of reliability. In *Proceedings of the Seventh IFIP Conference on High Performance Networking (HPN'97)*, April 1997.

[13] J. Magee, N. Dulay, and J. Kramer. Regis: A constructive development environment for parallel and distributed programs. *IEE Distirbuted Systems Engineering Journal*, 1(5):304–312, September 1994.

[14] B. Moon and J. H. Saltz. Scalability analysis of declustering methods for multidimensional range queries. *IEEE Transactions on Knowledge and Data Engineering*, April 1997.

[15] B. Mukherjee and K. Schwan. Improving performance by use of adaptive objects: Experimentation with a configurable multiprocessor thread package. In *Proc. of the Second International Symposium on High Performance Distributed Computing (HPDC-2)*, pages 59–66, July 1993.

[16] Object Management Group. The OMG web site. http:-//www.omg.org.

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

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

[19] D. M. Silva. COBS-OM performance evaluation. http://-www.ime.usp.br/ dilma/DataObject/, March 1998.

[20] 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.

[21] C. Zimmermann and V. Cahill. Open to sugestions: on adaptable, distributed application support architectures. In *European Research Seminar on Advances in Distributed Systems*, 1995.