How to Achieve Modularity in Distributed Object Allocation

Franco Zambonelli
Dipartimento di Scienze dell'Ingegneria Università di Modena
Via Campi 213/b – 41100 Modena – ITALY
Ph.: +39-59-378517 – Fax: +39-59-370040
E-mail: zambonelli@dsi.unimo.it

Abstract

The paper focuses on language constructs for driving the allocation of parallel object-oriented applications onto a target architecture. The paper analyses the issues that arise in the definition of these constructs and presents the solutions adopted in several systems and programming environments, by discussing their capability of enforcing the principle of modularity. Open issues and future directions of research are outlined.

Key-Words: Allocation, Dynamicity, Modularity, Object-Orientation, Language Constructs

1. Introduction

The allocation problem, i.e., the problem of a wise assignment of the components of a parallel application to the execution resources so to achieve high performance, is one of the key points to be faced in the design of any parallel and distributed programming environment [CasK88].

Fully automated tools can be implemented and integrated within a distributed environment. They are in charge of controlling the execution of a parallel application and deciding, on the basis of an automated **allocation policy**, the allocation of its components. The main advantage of automated allocation tools is to provide **transparency**: the user can disregard any issue related to allocation and leave all decisions to the policy. A drawback of an automated approach is that it is only capable of reaching general purpose goals and cannot to take into account the peculiar application needs: different applications can have different needs in terms of system resources, and the same allocation policy may have different impact on them.

An alternative approach may sacrifice transparency to give the user the capability of driving the allocation of a parallel application. Though such a solution may be viable for static applications, whose allocation can be effectively decided before their execution, dynamic applications exist whose allocation can be effectively decided only at run-time. Because a dynamic intervention of the user is highly intrusive and, generally, not tolerable, a customized dynamic allocation policy must be somehow codified within the application.

The paper restricts the focus to object-oriented parallel programming environments [Weg90, ChiC91] and analyses those **language constructs** that can be introduced to permit users to express the allocation behavior of their application components and to drive their allocation. Though very different approaches can be adopted, a few design issues can precisely characterize them, such as the degree of **architecture independence** of the constructs, their **expressive power**, their degree of integration with the **run-time system**. In addition, a primary requirement for the analyzed allocation constructs is the capability of **enforcing modularity** in the definition of the allocation behavior, i.e., the constructs must be confined and separated from other application parts and they must grant reuse of the defined allocation behavior via context independence.

The paper surveys and analyses the above issues, with attention to modularity, several object-oriented programming environments and systems that provide some kind of allocation constructs; among the others, the Parallel Objects environment [CorL91, CorLZ97], whose project personally involved the author. Without the ambition of defining a precise taxonomy, the paper outlines important similarities and differences among the facilities offered by different systems. In addition, the analysis permits to highlight open issues in the area and to suggest new directions of research.

The paper is organized as follows. Section 2 analyses the allocation problem in its general terms. Section 3 introduces the customized approach to allocation and section 4 its main design issues. Section 5 surveys the existing systems. Section 6 outlines open issues and future directions of research.

2. The Allocation Problem

A parallel application defines a set of logical components in need of execution and of interacting/communicating each other. The allocation problem, in its general terms, consists in finding one assignment of the application components to the available physical resources of the target architecture that achieve **high-performance**, i.e., a good speed-up depending on application characteristics and available resources [CasK88].

2.1 Static versus Dynamic Allocation

Based on the time at which allocation decision are taken, one can distinguish between static and dynamic approaches. In the **static approach**, the allocation of the components of a parallel application onto a target architecture is decided before the execution of the application itself. A static approach to allocation is suitable only for those applications whose behavior can be accurately predicted at compile-time [NorT93]. Whenever the behavior of an application is

dynamic, i.e., it cannot be predicted at compile-time, no effective allocation decision can be taken before the execution. In those cases, a **dynamic approach** must be adopted [ShiKS92]: the allocation of the components of a parallel application must be decided at run-time and adaptively, depending on the current state of the execution. One the one hand, the allocation of each component can be decided when it begins its active life within the system (i.e., when it is dynamically created). On the other hand, if a **migration** mechanism is available [Smi88, Nut94], the allocation of already allocated entities can be changed during their life.

2.2 Automated Allocation Tools

To manage the allocation of a parallel application onto a target parallel/distributed system is a complex activity, whether a static or a dynamic approach is adopted.

To provide full **transparency** and to free the user from any allocation-related issue, a parallel programming environment can integrate automated tools for dealing with either static or dynamic allocation (or both). In the static case, it is the job of the compiler to extract from the application code all the information needed and to map it onto the target architecture on the basis of a mapping algorithm [NorT93]. Static code analysis and off-line monitoring can produce a large amount of useful information about resource exploitation of the application components. In the dynamic case, the application behavior is monitored at run-time and then, on the basis of the obtained information, the allocation of the application components is decided by an allocation policy in charge of dynamically issuing the needed mechanisms [CorLZ92].

3. Toward User-Driven Allocation

Automated allocation tools have the great advantage of making the allocation transparent to the user. However, they have limits that can make it necessary to users the control of the allocation of their applications.

3.1 Limits of Automated Allocation Tools

Any automated approach to allocation fixes a general-purpose goal for the allocation policy. In the past few years, many research efforts have been devoted to the study of both static and dynamic allocation policy with the goal of either load sharing or load balancing [ShiKS92]. In addition, several systems and programming environments have integrated automated allocation tools based on a fixed load balancing/sharing policy [PowM83, BalK93]. However, all of the above proposals are more system-oriented rather than application-oriented: the same policy applied to all applications without any variation neglects peculiar application needs. For example, an automated policy could allocate on distant nodes, with the aim of load balancing, heavily communicating entities, making them incur in high communication costs, because the components of a parallel application usually needs to interact each other [Jul88]. Furthermore, by assigning the entities situated on the critical path of an application the same amount of execution resources assigned to the other application components can make the whole application execution time increase [ShiWP90].

Static allocation policies – because of their off-line and non intrusive nature – can somehow take into account, in their decisions, detailed information about the execution and the communication behavior of the application components. The situation is different in the dynamic case: it is too expansive to dynamically collect detailed information about the behavior of the application components and to take them into account in allocation decisions. Only a very limited amount of application-dependent information can be collected and exploited in decisions, because of the on-line and intrusive nature of dynamic allocation tools.

3.2 Explicit Allocation Code

When the performance achieved by the decisions of automated allocation tools is not satisfactory, one might decide to sacrifice transparency for performances. The user, in this case, has the duty of studying his/her applications' behavior and somehow driving their allocation.

The off-line nature of static allocation can make it quite simple to take into account the peculiar needs of an application: the user can fully manually specify, before the execution of an application, where its components must be allocated, even with the help of some graphical tool [Bru93]; in addition, the user can easily check the decisions taken by a mapping algorithm to tune its behavior.

To take into account the dynamic allocation needs of an application is more complex. The on-line nature of the decisions makes impossible to leave the user the duty of directly detecting the applications' behavior and deciding the allocation of their components at run-time: this interaction would produce an intolerable overhead and the user cannot be prompt enough to make his/her decisions effective.

A different approach is necessary to allow the execution of a parallel application to be dynamically managed with regard to its peculiar allocation needs: a parallel application must contain in itself not only the "algorithmic" code but also some kind of "**allocation code**". The allocation code must allow users to specify the allocation needs of their application components or, in other words, must help in building a customized dynamic allocation policy.

The capability of controlling the allocation of the application component from the application level can be provided by the run-time support to the environment, that gives users access to the low-level allocation mechanisms. This capability can be given in the form of:

- **library calls**, with a well defined interface, that give direct access to the allocation mechanisms available in the environment [Zhou93, Tar94];
- **language constructs** whose execution has the effect of influencing the allocation of the application onto the architecture [Jul88, Ach93].

In both cases, the allocation code is no longer transparent to the programmers that acquire, by means of the allocation code, the capability of building their own dynamic allocation policies, tuned to the peculiar needs of their applications.

The above distinction between library calls and language constructs is not the most significant one: in many cases, a

sequence of language constructs is simply translated by the compiler in a sequence of library calls to the run-time support. Thus, the following of this paper focuese on language constructs only.

4. Design Issues in the Allocation Code

Very different approaches can be adopted in the definition of language constructs to provide application-level access to the allocation mechanisms. This section outlines and analyzes the issues that characterize most significantly a given proposal in the area, with a peculiar attention to modularity.

4.1 Modularity

The modularity principle enforces separation of concerns and tries to build encapsulated and reusable modules. In the allocation area, modularity expresses the capability of:

- **separating** the allocation code from the algorithmic code of an application;
- **confining** the area of influence of the allocation code for a specific application or component of an application.

With regard to the first point, the code that implements the core functionality of an application has a goal different from the code that implements the allocation policy. Hence, if separation of concerns is not enforced and, instead, different kinds of code are mixed together, the complexity of an application is likely to increase, making it more difficult design and maintenance. The same separation principle is encouraged, for example, in the area of synchronization for concurrent programming [CorL91, McH94].

The second point relates to reuse, that can be granted only by enforcing encapsulation and context-independence. Because the allocation code associated to an application or to the components of an application becomes part of the behavioral description of the components, one must avoid the allocation behavior of one component to be influenced by other external entities: this would be a violation of the encapsulation principle and would no longer guarantee the same expected behavior in different contexts. Conversely, the allocation code associated to a given component must not influence the allocation of any external entity.

4.2 Other Design Issues

Though the enforcement of the modularity principle is a primary requirement, other important issues characterize language constructs for allocation. In particular:

- **abstraction**, i.e., the degree of architecture independence the allocation code expresses and, consequently, the capability of making applications portable;
- **expressive power**, i.e., the capability of the allocation code of giving user full control over the allocation of his/her application components;
- **integration with the run-time support**, i.e., the degree of control the run-time support has over the allocation and the way it interacts with the user-defined allocation code.

The lack of a dominant architecture makes architecture-independence and portability a general requirement in the parallel computing area. The allocation code must answer them too, by expressing the allocation needs of parallel applications in abstract and architecture-independent terms that are likely to assume different concrete meanings on different architectures, transparently to the user. A drawback of an architecture-independent approach is that it cannot always grant a very efficient exploitation of the resources of the target architecture, because of the introduced abstraction layer. With regard to modularity, the application of a modular principle in the definition of the allocation code and architecture-dependency seem to clash: the reusability degree of the application components would be too limited by granting portability only across different application but not across different architectures.

By recalling that the reason for writing application-specific allocation code is that a single policy in the run-time system is not be suitable for all applications, an environment that provides tools for writing application-specific policies must grant tools powerful enough to express a wide range of allocation policies. In the allocation area, it can be very difficult to understand the power of either a set of language construct or of library calls. Expressiveness can be measured by the amount of information about the state of the system that can be exploited, the available allocation mechanisms and the capacity of directly commanding them. The respect of the modular principle intrinsically limits the expressiveness of the allocation code; nevertheless, we claim this limit is essential to keep low the complexity of the allocation code.

With regard to the run-time system, the starting point of this paper is that a user-defined allocation policy is likely to perform better than a general-purpose allocation policy. However, we feel that leaving to the user the duty of managing the allocation of its parallel application from scratch and without any help from the run-time system may not to be the optimal solution. An alternative approach can integrate an automated allocation policy and some kind of allocation code, conceived with the purpose of customizing its automated behavior. An important advantage that can come from this integration is that the user is no longer obliged to specify the allocation code: if no allocation code is defined, the system policy performs anyway some dynamic management on the allocation of the parallel application on the system. An apparent drawback of an integrated approach is that it seems to clash with modularity, by allowing an external entity – the automated policy – to influence the allocation of the application components. Despite of that, a clear separation of concerns is still enforced: the allocation code can be defined only on the basis of application-level information; the policies implemented in the run-time system take allocation decisions based on low-level load information about the current state of the system. Only by allowing an external entity to exploit this low-level information one can grant architecture-independence and ease of use without loosing in efficiency.

5. Survey of Existing Environments

Several proposals in the concurrent object-oriented area

aim to define language constructs for customizing the allocation of objects in parallel/distributed applications. As a general rule, all proposals make the allocation code become part of the definition of a class (figure 1), as it has been for language constructs introduced for different purposes, such as synchronization and deadline specification [CorL91, Aks94]. In fact, because the application components are instances of a class that describes their behavior, the dynamic allocation needs of the components have to be considered as part of the behavior. In addition, all proposals focus on locality of references between objects as the key issue to achieve high-performance. Apart from this common characteristics, very different approaches have been followed, as the following sub-sections show.

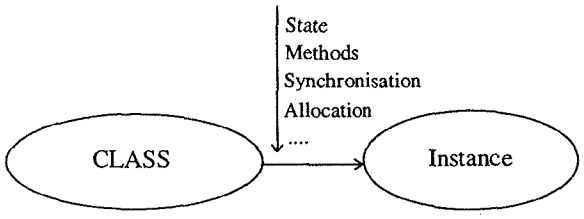

Figure 1. The allocation code as part of a class definition

5.1 Emerald and Friends

Emerald [Jul88] is one of the systems that firstly introduced language-level constructs to dynamically allocate the objects of a parallel applications.

Emerald can insert constructs for allocation either as instructions in the code of the instance methods or as annotations in the definition of an instance variable (see figure 2).

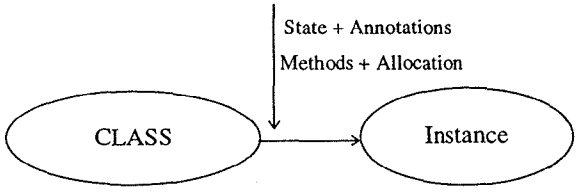

Figure 2. The Emerald approach

With regard to the former case, Emerald defines several constructs for object mobility (i.e., migration), that have the effect of making the corresponding mechanism apply. The move construct can be used to migrate an object to a given location, i.e., to the node of residence of another object. For example, the instruction:

```
move X to locate Y
```

provides, when executed, to move the object referred by the variable X to the node where the object referred by Y currently resides. The move primitive can also be used in parameter-passing when invoking an operation upon an object. For example, the following statement:

```
foo.SomeOperation(move X)
```

specifies the compiler that the parameter object (in this case X) should be migrated to the node where the object referred by foo is located. In Emerald parlance this is known as *call-by-move*. Another keyword, visit, can be used instead of move in parameter passing: in this case, the parameter is to be migrated to the destination node for the duration of the call and moved back upon termination. Call-by-move and call-by-visit are provided in the language both as a convenience to the programmer (they avoid the need for explicit move statements before a call) and as a performance optimization (they permit the compiler to pack the migrating objects in the same network packet of the invocation).

The last mobility concept that Emerald supports is *fixing*, which requires an object to become immobile. If one object is fixed at a node it cannot be migrated to another node, unless it is later unfixed and thus made mobile again or refixed to a different node. One reason why one object may be fixed at a node is if it is currently accessing resources specific to that node. The syntax used in Emerald for fixing, unfixing and refixing are as follows:

```
fix X at Y
unfix X
refix X at Y
```

Any move and visit commands are ignored by the run-time system if applied to one object which is currently fixed.

Objects do not live in isolation but contain references to other objects, e.g., an object may be the root node of a graph or a linked list. When such an object is being moved to another node, it may be appropriate to move the entire graph/linked list, so to grant locality. The attach annotation, used in the definition of instance variables, is provided for this purpose. For example, the following definition:

```
attach var foo: type_of_foo
```

specifies that the foo instance should always follow the movements of the objects in which it is declared. Whenever the object into which this declaration is contained migrates, it requires the migration of any object eventually attached to foo, in a transitive way.

Emerald constructs are very powerful and make it possible to express a great variety of allocation behavior in applications. In addition, they are abstract and portable: the allocation code of the application components is expressed in terms of the allocation of other components, without forcing (though allowing) the user to refer, say, to a physical node of the system. In spite of these advantages, the Emerald way of dealing with the allocation of application components is far from enforcing the modular principle. Firstly, allocation constructs are mixed with the sequential code (methods and instance variables definitions). Thus, no separation of concerns is provided and the user is forced to define his/her classes by taking into account, at the same time, algorithmic and allocation-related issues: this is likely to increase the complexity of parallel applications. In addition, the allocation constructs may tend to violate the encapsulation principle. In fact, the constructs inserted in the code of a class do not

specify the allocation behavior of the instances of the class itself (i.e., of the object that is currently executing them) but they can command the migration of other instances of which the reference is hold. This happens, for example, because an object can be moved around in the system commanded by the clients that are currently refereeing it or by objects attached to them.

As a final remark, in Emerald all responsibility of allocation is left to the user and there is neither any system-level allocation policy to check or integrate the user-inserted allocation constructs nor any way to access to information about the current system state (for example, the load of the nodes). Only a few static allocation decisions can be autonomously taken at compile time: for example, the compiler generates code to move a parameter if it is a small and immutable object, e.g., an integer, because such objects can be copied unexpensively.

The Emerald approach has influenced many other object-based systems such as Amber, Distributed Smalltalk and Trellis/DOWL.

Amber [Cha89] shapes its object mobility primitives after those of Emerald. It offers facilities to move an object via the MoveTo primitive, to Attach one object to another and to Unattach an attached object. A peculiar characteristic of Amber is the capability of marking an object as *immutable*: in this case, invoking MoveTo on it will *replicate* the object rather than move it, thus allowing better availability without incurring in any consistency problem.

Distributed Smalltalk [Ben90] provides the move and copy instructions to *migrate* and *replicate* an object on a given node, respectively. In addition, it provides the ability to optionally state the node at which an object is to be created. Again, the syntax is very similar to the Emerald's one.

The **Trellis/DOWL** system [HeuA89, Ach93] offers the same allocation constructs of the Emerald system and, again, mix them in the algorithmic code and do not integrate them with any automated allocated policy. However, the semantic of the constructs differ from that of Emerald, and represents a step toward encapsulation. In fact, allocation constructs are introduced to specify the dynamic allocation behavior of the object onto which the constructs are inserted, without influencing the allocation of other application objects, say of objects to which the reference is held.

For instance, in Trellis each object has instance variables called location and fixed_at. Assigning a value to these variables cause an object to migrate to, or to become fixed at, a location: since instance variables can be assigned only from the methods of the objects itself, no other components of an application can decide of migrating or fixing an object to a given node. Analogously, the attachment is expressed by means of an instance variable that specifies (in the form of a head and tail list), to which other objects an object must be attached: differently from Emerald, it is an objects that declare its attachment to another one, and so deciding of following it in its migration, not viceversa.

The move and visit primitives of Trellis serve basically the same purpose as they do in Emerald but, again, they are used in a different way: they move and visit primitives do not appear in the *invocation* of an operation, as in Emerald, but in its *declaration*. For example:

```
operation foo(a:visit someType, b:move otherType)
```

The above declaration makes the actual parameter of an invocation of the foo operation migrate. Though the parameter objects are still influenced in their allocation by external entities (i.e., the invoking object), the allocation behavior is somehow a bit more context-independent, because it is the same for any invocation of the same operation.

5.2 Reflective Systems

Reflective systems are characterized by the introduction a **meta computing level** in which is stored and processed data to model the computation [WatY88]. Reflective systems delegate any issue related to the management and the optimization of application execution to the meta-level that is capable, on the basis of the current state of the computation, of influencing the behavior of applications. Programming a reflective system means not only to write the usual algorithmic code (the so called base-level) but also to write the meta-level code: in that way, the algorithmic part of the code is completely separated from its management part.

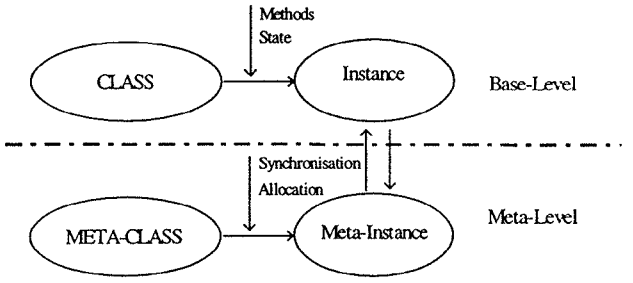

Figure 3. Allocation in an Object-Based Reflective System

In concurrent object-based reflective systems [WatY88], each base-level object is associated to a meta-level object, in charge of managing its communication and synchronization. A natural extension is to give meta-level objects the responsibility of allocation too [Mas94, OkaI94, Lux95], such as locality of reference, load balancing, management of prioritized scheduling policies (see figure 3). When programming the meta-level, the user has access to all the information usually exploited by automated allocation tools, such as the state of hardware resources, the current load of the nodes, the location of the objects in the systems. In addition, by the meta-level, the user can directly control the location of the base-level objects by commanding their migrations. This allows the user to write his/her own allocation policies for the objects of the system.

In the **AL/1D** system [OkaI94], for example, meta-level objects can store, as instance variables, load information about the state of the system nodes and about the location of other objects in the system. Operations of the meta-level objects, triggered by the events occurring at the base-level, can access this variables and issue the allocation mechanisms to implement specific allocation policies. As an example, a method of the Foo class can be defined in the meta-object to

react to any message send action and to provide the migration of the sender toward the receiver, whenever the receiver is the object bar. This is shown, with the AL-1D syntax, in the below code:

```
meta Foo/* declaration of the meta-object */
vars localHost; /* meta level instance variable */
     /* node of residence of the base-level object */
.....
method Foo ssend: rcvr: msg
/* method to react at base-level send events */
vars rcvr_location; /* local variable */
     rcvr_location = NameServer location: rcvr
     /* find the location of the receiver */
     if (receiver_location != localHost AND
          (rcvr name) == #bar) then
           state migrating: rcvrLocation
     .....
```

The **Distributed Memory Reflective Architecture** system [Mas94] enlarge the meta-level capabilities by introducing different meta-level objects apart from the one associated with the base-level objects of the application. For example, a meta-level object called node manager can be introduced to describe the computational state of one node of the system and can collectively manage all base-level objects allocated in the node; In addition, a meta-level object called class manager can be associated to each class of the system to manage the resources shared by all the objects within a class. In some sense, these additional meta-level objects can assume the role of an allocation policy within the run-time system and have the capability of taking allocation decisions based on system-state information not easily accessible at the level of single instances.

A well confined problem is faced in the **BirliX** system [Lux95]: not only the allocation policy for a given object can be customized, but the migration mechanism too, to allow grater flexibility and efficiency.

Due to the introduced meta-level, reflective systems provide a clear separation of concerns (the allocation code is confined and not mixed with the algorithmic code) and encapsulation (the meta-level influences the allocation of the associated instance only). Then, they represent an important evolution toward the respect of a modular principle in the allocation code. In addition, reflective system gives users full power in programming customized allocation policies for their application: they permit to exploit system-level information to specify the allocation behavior for both single instances and groups of objects. In spite of this flexibility, the definition of the meta-level is prone to become low-level and, then, less portable and difficult to be written (as depicted from figure 3).

5.3 Parallel Objects

The Parallel Objects programming environment [CorL91], based on the active object model [ChiC91], addresses the allocation problem by integrating a system-level allocation policy with a set of *high-level directives* (collectively called Abstract Configuration Language, shortly ACL) to specify the peculiar allocation needs of application objects [CorLZ97].

The load balancing policy in the run-time support automatically decides about the allocation of newly created objects and about the migration of already allocated one, depending on the current system load [CorLZ92]. Though capable of autonomous decisions, the policy is forced to act in agreement with the ACL directives, and then adapts its behavior to the peculiar allocation needs of an application (see figure 4). Depending on the user skill with the allocation problem, ACL directives could be more or less constrainable. At one extreme, the user can totally ignore them and let the allocation to be completely decided by the system policy; since the system policy cannot always act respecting the application needs, this might produce inefficiency. At the other extreme, the user can specify the application allocation needs in a complete way as to almost (or fully) disable the automatic allocation policy; if the directives specified by the user are clever, they will realize "ad-hoc" allocation policies for his/her applications.

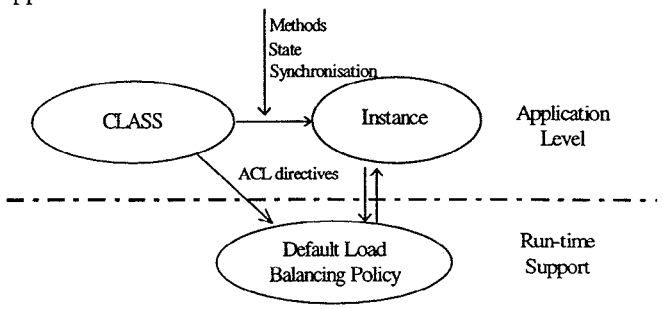

Figure 4. ACL directives interact with the PO run-time support

ACL directives may refer either to general allocation properties of one objects and of its internal components (in Parallel Objects, one objects is not considered a whole unit of allocation but, instead, its state and the activities devoted to the services of the requests can be distributed [CorLZ97]) or the allocation of one object in relation with other application objects. Examples of ACL directives are:

- distributed(#nodes): allows one object to distribute its components onto several nodes;
- migratable(): allows one objects to be migrated around the system;
- close_to(O1): specifies the need of one object to be allocated near the object O1;
- neighbour(O2): specifies the need of one object to be allocated near the object O2, but not on the same node, so to allow them to execute in parallel without competing for execution resources.

It is important to note that ACL directives dynamic requirements can be parameterized and related to the current state of the object. For example, one ACL directive can specify the locality need of the instances of a class in relationship with an instance whose reference is maintained in an instance variable. The fact that a reference to one object con change in time, makes the behavior specified by the directive dynamically change.

ACL directives are abstract and portable, because they

express logical rather than physical concepts related to the allocation of the components. Only at compile-time these concepts assume concrete meaning, depending on the particular characteristics of the target architecture. For examples, the "closeness" concept can assume the meaning of "coresidence" in a distributed environment, "on the same cluster" in a clustered environment.

With regard to modularity, ACL directives enforce both separation of concerns and encapsulation. One the one hand, they are specified in a special section of the class definition, as follows:

```
PO_CLASS foo
    STATE_section
        /* state definition */
    METHODS_section
        /* methods definition */
    ACL_section
        /* list of ACL directives */
END_CLASS foo
```

On the other hand, even if ACL specifies the allocation needs of an objects in terms of the allocation of other objects, the run-time support does not interpret them as transitive and does not perform any allocation action on one object unless it agrees with the ACL directives associated to the object.

A limit of the Parallel Objects approach with respect to Emerald-like and Reflective system is the limited expressive power that high-level directives induce. However, to our opinion, this lack of expressiveness is somehow supplied by the automated decisions of the integrated load balancing policy.

6. Open Issues and Future Work

The definition and the implementation of language constructs to customize the allocation of parallel application is a quite new area of research. From the analysis of the above proposals, we identified several open issues and promising areas of research.

First of all, it is necessary a deep analysis of the performance benefits provided by these constructs. If wise application of allocation code can potentially increase performances, conversely it could also be used inappropriately and, then, it can degrade performance. For example, migrating an object close to the server it is currently accessing can increase performance only if the number of accesses is high enough to outweigh the migration cost; otherwise, that will degrade performances. Currently, none of the systems surveyed in the paper introduces tools to evaluate the effectiveness of the allocation code specified within an application. In addition, though the application of the modularity principle and the abstraction of the language constructs can make it theoretically portable, this must also be supported by performance portability. All the surveyed systems claim portability, but none of them have proved their allocation code to be effectively portable – in terms of performances – onto different architectures. In this direction, formal methods could help in the definition of the allocation code and in its performance analysis

A second area that needs to be analyzed relates to reusability and comes from the inheritance anomaly problem [McH94]. Inheritance involves not just code *reuse* but also code *change*; if a class contains two kinds of code – for example sequential code and synchronization code – a *change* to one of these kinds of inherited code can hinder the *reuse* of the other kind of inherited code (and viceversa). In the worst case, a subclass may not be able to reuse *any* inherited code. Most of the analysis of the inheritance anomaly have focused the domain of synchronization, but the inheritance anomaly may appear whenever one inherits two or more kinds of code within a hierarchy. For example, inheritance anomalies have been reported even in the area of object-oriented real-time systems with regard to deadline specifications [Aks94]. Thus, the problem is likely to represent itself in the analyzed domain of allocation [Mas94]. However, to our knowledge, no specific analysis of the problem exists and little is known about how serious the problem can be.

A further interesting area of research concerns the strict relation between the allocation code and the synchronization code. The area of allocation seems to overlap with the area of synchronization for concurrent object-oriented languages in several important ways:

- the same event-based programming model used in several synchronization proposal [CorL91, McH94], can be used for the allocation. This event-based model has been explicitly identified in reflective systems, but characterizes also Emerald-like systems (the invocation of a method can cause its parameter to migrate) and Parallel Objects (the creation of an objects starts a decisional activity within the integrated load balancing policy);
- the semantics of the accesses to a given component of an application (and the way they are synchronized) is likely to provide important hints with regard to its allocation. For example, if a component is accessed in mutual exclusion, it could be migrated to the component that is currently accessing so to enforce locality of reference and with the guarantee that no other local access will be broken. As a further example, one object with a read-only state can be replicated without any consistency problem.

The overlap between allocation and synchronization suggests that researchers in the relatively young area of allocation might be able to derive some useful ideas from the deeper analyzed area of synchronization. With regard to the modular issue, we suggest to consider that the allocation area is evolving in a very similar to the one of synchronization: as the latter evolved from semaphores up to monitor and path-expression, the allocation area is evolving from low-level constructs mixed in the algorithmic code (Emerald), up to meta-level allocation objects (Reflective Systems) and high-level directives with a well-defined scope and separated from the algorithmic code (Parallel Objects).

7. Conclusions

The paper surveys several object-oriented programming environments that provide language constructs to express the dynamic allocation behavior of applications: among a variety

of heterogeneous approaches, a few issues can characterize them. The paper focuses on modularity and analyses whether and how the introduced constructs permit to follow a modular principle in the allocation code.

Promising areas of research are offered by the relationship of the allocation problem with the synchronization area and by the analysis of the inheritance anomaly in the allocation code.

8. References

[Ach93] B. Achauer, "The DOWL Distributed Object-Oriented Language", Communications of the ACM, Vol. 36, No. 9, pp. 48-55, Sept. 1993.

[Aks94] M. Aksit et al., "Real-Time Specification Inheritance Anomalies and Real-Time Filters", Proceedings of ECOOP '94, LNCS No. 821, pp. 386-407, Springer-Verlag, July 1994.

[BalK93] H. E. Bal, M. F. Kaashoek, "Object Distribution in Orca using Compile-Time and Run-Time Techniques", Proceedings of OOPSLA '93, ACM SigPlan Notices, Vol. 28, No. 10, Oct. 1993.

[Ben90] J. Bennet, "Experience with Distributed Smalltalk", Software: Practice and Experience, Vol. 20, No. 2, pp. 157-180, Feb. 1990.

[Bru93] D. Bruschi et al., "A User-Friendly Environment for Parallel Programming", Proceedings of the 1st EUROMICRO Workshop on Parallel and Distributed Processing, IEEE CS Press, pp. 451-456, Jan. 1993.

[CasK88] T. L. Casavant, J. G. Kuhl, "A Taxonomy of Scheduling in General-Purpose Distributed Computing System", IEEE Transactions on Software Engineering, Vol. 8, No. 4, pp. 141-154, Feb. 1988.

[Cha89] J. S. Chase et al., "The Amber System: Parallel Programming on a Network of Multiprocessors", Proceedings of the 12th ACM Symposium on Operating Systems Principles, ACM Operating Systems Review, Vol. 23, No. 12, pp. 147-158, Dec. 1989.

[ChiC91] R. S. Chin, S.T. Chanson, "Distributed Object-Based Programming Systems", ACM Computing Surveys, Vol. 23, No. 1, pp. 91-124, March 1991.

[CorL91] A. Corradi, L. Leonardi, "PO Constraints as Tools to Synchronize Active Objects", The Journal of Object-Oriented Programming, Vol. 4, No. 6, pp. 41-53, Oct. 1991.

[CorLZ92] A. Corradi, L. Leonardi, F. Zambonelli, "Load Balancing Strategies for Massively Parallel Architectures", Parallel Processing Letters, Vol. 2, No. 2 & 3, pp. 139-148, Sept. 1992.

[CorLZ97] A. Corradi, L. Leonardi, F. Zambonelli, "High-Level Directives to Drive the Allocation of Parallel Object-Oriented Applications", Workshop on High-Level Programming Models and Supportive Environments, IEEE CS Press, Geneva (CH), April 1997.

[HeuA89] L. Heuser, B. Achauer, "Language Constructs to Express Distribution of Object-Oriented Applications", Proceedings of TOOLS '89, Paris (F), Nov. 1989.

[Jul88] E. Jul et al., "Fine Grained Mobility in the Emerald System", ACM Transactions on Computer Systems, Vol. 6, No. 1, pp. 109-133, Feb. 1988.

[Lux95] W. Lux, "Adaptable Object Migration: Concept and Implementation", ACM Operating Systems Review, Vol. 29, No. 5, pp. 54-69, May 1995.

[Mas94] H. Masuhara, "Study on a Reflective Architecture to Provide Efficient Dynamic Resource Management for Highly Parallel Object-Oriented Applications", Master Thesis, University of Tokyo, Tokyo (J), Feb. 1994.

[McH94] C. Mc Hale, "Synchronisation in Concurrent Object-Oriented Lanaguages: Expressive Power, Genericity and Inheritance", PhD Thesis, Department of Computer Science, Trinity College, Dublin (IR), Oct. 1994.

[NorT93] M. G. Norman, P. Thanisch, "Models of Machines and Computation for Mapping in Multicomputers", ACM Computing Surveys, Vol. 25, No. 3, pp. 263-302, Sept. 1993.

[Nut94] D. Nuttel, "A Brief Survey of Systems Providing Process or Object Migration Facilities", ACM Operating Systems Review, Vol. 28, No. 4, pp. 64-79, Oct. 1994.

[Oka94] H. Okamura, Y. Ishikawa, "Object Location Control Using Meta-level Programming", Proceedings of ECOOP '94, LNCS No. 821, pp. 299-319, Springer-Verlag, July 1994.

[PowM83] M. L. Powell, B. P. Miller, "Process Migration in DEMOS/MP", Proceedings of the 9th ACM Symposium on Operating Systems Principles, ACM Operating Systems Review, Vol. 17, No. 5, pp. 110-118, May 1983.

[ShiKS92] N. G. Shivaratri, P. Krueger, M. Singhal, "Load Distributing for Locally Distributed System", IEEE Computer, Vol. 25, No. 12, pp. 33-44, Dec. 1992.

[ShiWP90] B. Shirazi, M. Wang, G. Pathak, "Analysis and Evaluation of Heuristic Method for Static Task Scheduling", The Journal of Parallel and Distributed Computing, Vol. 10, No. 3, pp. 222-232, March 1990.

[Smi88] J.M.Smith, "A Survey of Process Migration Mechanisms", ACM Operating Systems Review, Vol. 22, No. 3, pp. 28-40, July 1988.

[Tar94] E. Tarnvik, "Dynamo: a Portable Tool for Dynamic Load Balancing on Distributed Memory Multicomputers", Concurrency: Practice and Experience", Vol. 6, No. 8, pp. 613-639, Dec. 1994.

[WatY88] T. Watanabe, A. Yonezawa, "Reflection in an Object-Oriented Concurrent Language", Proceedings of OOPSLA '88, ACM SigPlan Notices, Vol. 23, No. 11, pp. 306-315, Nov. 1988.

[Weg90] P. Wegner, "Concepts and Paradigms of Object Oriented Programming", ACM OOPS Messenger, Vol. 1, No. 1, pp. 7-87, Aug. 1990.

[Zhou93] S. Zhou et al., "Utopia: a Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems", Software Practice and Experience, Vol. 32, No. 12, pp. 1305-1336, Dec. 1993.