into the paradigm of object-oriented programming. A group may consist of objects and may be viewed as an object. Group communication may be expressed by means of object invocations. Programming with groups does not violate premises of object-oriented programming but strengthens them in a number of ways. Groups add to encapsulation because they hide complex interactions. Groups and group communication allow to abstract from such issues as communication protocols, membership, and ordering of events. Group mechanisms increase reusability by providing a framework for reuse of structure, patterns of interactions.

Group communication and group mechanisms have been recognized as an adequate tool for distributed systems and for fault tolerant computing. We believe that also object-oriented programming may significantly benefit from application of these concepts.

Acknowledgements

I would like to thank Prof. Krzysztof Zieliński for help in clarification of some concepts. I am also grateful to Grzegorz Czajkowski and Bjarne Steensgaard for their comments on drafts of this paper.

References

[1] Kenneth P. Birman. The Process Group Approach to Reliable Distributed Computing. Technical Report 91–1216, Cornell University, July 1991. To appear in Communications of the ACM.

[2] Andrew P. Black and Mark P. Immel. Encapsulating Plurality. unpublished, December 1992.

[3] David R. Cheriton and Willy Zwaenepoel. Distributed Process Groups in the V Kernel. *ACM Transactions on Computer Systems*, 3(2):77–107, May 1985.

[4] Eric C. Cooper. Replicated Procedure Call. *Operating Systems Review*, 20(1):44–56, January 1986.

[5] Michael J. Fisher, Nancy A. Lynch, and Michael S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, April 1985.

[6] Olof Hagsand, Holger Herzog, Kenneth Birman, and Robert Cooper. Object-oriented Reliable Distributed Programming. In *Proceedings of the 2nd International Workshop on Object-Orientation in Operating Systems IWOOOS'92*, Dourdan, France, September 1992.

[7] M. Frans Kaashoek and Andrew S. Tanenbaum. Group Communication in the Amoeba Distributed Operating System. In *Proceedings of the 11th International Conference on Distributed Computer Systems*, pages 222–230, Arlington, VA, May 1991.

[8] Pattie Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of the 1987 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 147–155, October 1987.

[9] Mesaac Makpangou, Yvon Gourhant, and Mark Shapiro. BOAR: A Library of Fragmented Object Types for Distributed Abstractions. In *Proceedings of the International Workshop on Object Orientation in Operating Systems*, pages 164–172, October 1991.

[10] Przemysław Pardyak. Group Communication in an Object-Based Environment. In *Proceedings of the 2nd International Workshop on Object Orientation in Operating Systems IWOOOS'92*, pages 106–116, September 1992.

[11] Rajendra K. Raj, Ewan D. Tempero, Henry M. Levy, Andrew P. Black, Norman C. Hutchinson, and Eric Jul. Emerald: A General-Purpose Programming Language. *Software — Practice and Experience*, 21(1):91–118, January 1991.

[12] Mahadev Satanarayanan and Ellen H. Siegel. Parallel Communication in a Large Distributed Environment. *IEEE Transactions on Computers*, 39(3):328–348, March 1990.

bly remote, processes. Also the first step of our research [10] gave a method for multicast invocation of objects. In contrast to these approaches, the mechanisms presented in this article give not only a method of group communication but a complete framework for creation, maintenance, and use of flexible groups.

Another approach to providing group communication is to create libraries that abstract operating system features. BOAR [9] is a collection of C++ classes that represent mechanisms available in the SOS system. Also group mechanisms in the ISIS system have been encapsulated in a hierarchy of classes [6]. Both libraries provide a user with a number of abstractions that mask operating system mechanisms. They are not a general language construct because their semantics depends on the system they are based on. Our mechanism could be ported to any object-oriented language and may use any kind of underlying operating system primitives.

There is an independent research done on group mechanisms in Emerald [2]. A gaggle, the proposed concept, is a very simple group mechanism without support for group communication. Objects may be added to a gaggle and when a gaggle is invoked then the invocation is passed to one of the gaggle's members. Our new group mechanism allows trivially to simulate gaggles but such simulation would be impossible in the opposite direction.

7 Next step

In our research we came to the point where the language mechanisms available in traditional object-oriented languages become the limiting factor. The problem is that we are reasoning about language aspects (like, for example, what happens when an object is being invoked) at the level of the language. We believe that the next step in the research towards specification of groups should be done in a reflective language.

A language is reflective [8] if it gives programs written in it access to their representation and permits them performing actions dependent on the representation. A group mechanism may be viewed as an ad-hoc reflective computation because it defines *meaning of an invocation*. This can be seen in the specification of the coordinator objects in our enhancements to Emerald where we define what happens when a corresponding group object is invoked. In this way a coordinator object is a *meta-object* for a group object. Also specification of protocols for invocation and result handling may be done more adequately if an invocation message is a first class object of the language. This feature is available in some reflective object-oriented languages due to the mechanism of *message reification*.

8 Conclusions

In this article, we presented group mechanisms for an object-oriented language. We created the mechanisms according to the proposed general model of a group and taking into account the identified requirements posed by the paradigm of object-oriented programming. We have implemented the mechanism in the Emerald language.

We believe that the notion of a group fits well

```
const NameServiceControllern ==
   group NameServiceController
      % operations available on the group
      export Add, Remove, LookUp, GetServers
      % protocols used for group maintenance
      call Broadcast.Total, Broadcast.Functional

      % an operation turned into MOI on members
      invocation Add[Name : String, Item : Any]
         % use atomic causal broadcast protocol
         call Broadcast.Total

         % called when members invoked
         init Receivers : Vector.Of[MemberType]
            % abort MOI when no members in the group
            quit when Receivers.Empty
         end init

         % called when a member returns
         reply received
            last isLast : Boolean

            % stop when all members returned
            quit when isLast
         end reply
      end Add

      invocation Remove[Name : String, Item : Any]
         ...
      end Remove

      % an invocation called on a single member
      invocation LookUp[Name : String] → [Item : Any]
         % use functional broadcast (invoke one member)
         call Broadcast.Functional

         % wait for the first result and return
         reply Result : Any received
            Item ← Result
         end reply
      end LookUp

      % operation performed locally on the coordinator
      function GetServers →
                  [Members : Vector.Of[MemberType]]
         % call the associated group object
         Members ← group.GetMembers
      end GetServers
   end NameServiceController
```

Figure 10: Example of a group coordinator

designed as parts of a group may be used in group invocations without any change.

```
operation Remove[Name : String, Item : Any]
   by Receivers : Array.Of[Any]
   if Receivers == NIL then
      % simple invocation
   else
      % MOI invocation
   end if
end Remove
```

Figure 11: Example of a use of the **by** clause

5.6 Summary of implementation

The Emerald implementation of groups fulfills our requirements for such a mechanism. A group is an object, group communication is achieved by means of object invocation. Groups are homogeneous and transparent. Coordinator permit flexible handling of MOI. The group mechanism does not interfere with mobility and is type-secure.

6 Comparisons

The success of group communication in distributed systems (e.g., V [3] and Amoeba [7]) provoked research into adopting the concept as a programming technique. In this section, we show some example approaches and compare them to our proposal.

The concept of a Remote Procedure Call have been enhanced to handle group communication. Replicated Procedure Call [4] allows to structure systems into troupes of replicated servers and serves as a procedure call among such troupes. The Multi-RPC [12] is a parallel RPC mechanism that allows one process to invoke a procedure in a number of other, possi-

```
% creation of a group
NSCreator ← Group.Of[NSServiceType, NSServerType]
NSGroup ← NSCreator.Create[NSCoordinator]
% building a group
NSGroup.Join[Server1]
...
NSGroup.Join[Server1]
NSGroup.Leave[Server2]
NSGroup.AddMonitor[NSMonitor]
% client's view of a group
NameService ← NSGroup.GetGroup
% using a group
NameService.Add["sirpa", Sirpa13]
NameService.Remove["foo", FooBar]
foo ← NameService.LookUp["sirpa"]
```

Figure 9: Construction and use of a group

lowing protocols:

- totally ordered reliable broadcast
- causally ordered reliable broadcast
- atomically ordered reliable broadcast
- unordered reliable broadcast
- functional call (one member is selected, if it cannot be reached another one is selected until success or exhausting possibilities),
- n-call (the same as above but we require invocation of n objects)

We use exclusively reliable protocols because the semantics of object invocation in Emerald is reliable: an invocation succeeds or an object is detected as unavailable. We preserve this semantics in our group mechanisms.

All protocols to be used in a group are defined in its coordinator. The **call** clause in the header of the coordinator's definition gives the two protocols to be used for maintenance. The same clause used in a definition of the **invocation** operation defines the protocol used for this particular operation. All messages pertaining to a group (group communication and management) are ordered relatively to each other.

5.4 Group communication

Group communication by means of Multiple Object Invocation is achieved by an **invocation** operation on a group coordinator. Each invocation of a group is turned into an invocation of its coordinator. If the invoked operation is a traditional one then it is normally invoked on the coordinator object. It it is an **invocation** operation then it is turned into invocations on members according to the protocol specified in the **call** clause. The body of the operation specifies reaction to subsequent events related to the course of MOI. Three parts of the body permit reaction to three kinds of events: starting an invocation, invocation of members, and reception of a result. The set of actually invoked members is made available.

5.5 Receiving broadcast invocation

An optional clause **by** has been added to a method definition syntax (Figure 11) to enable detection of MOI invocation and to reveal information otherwise unavailable at call time. A variable declared in this clause contains a list of all objects that received the same invocation message. If the invocation was not broadcast then the variable references the *NIL* object. If the list of receivers is not needed then the **by** clause may be omitted—this means that MOI mechanism is transparent on the callee's side: objects initially not

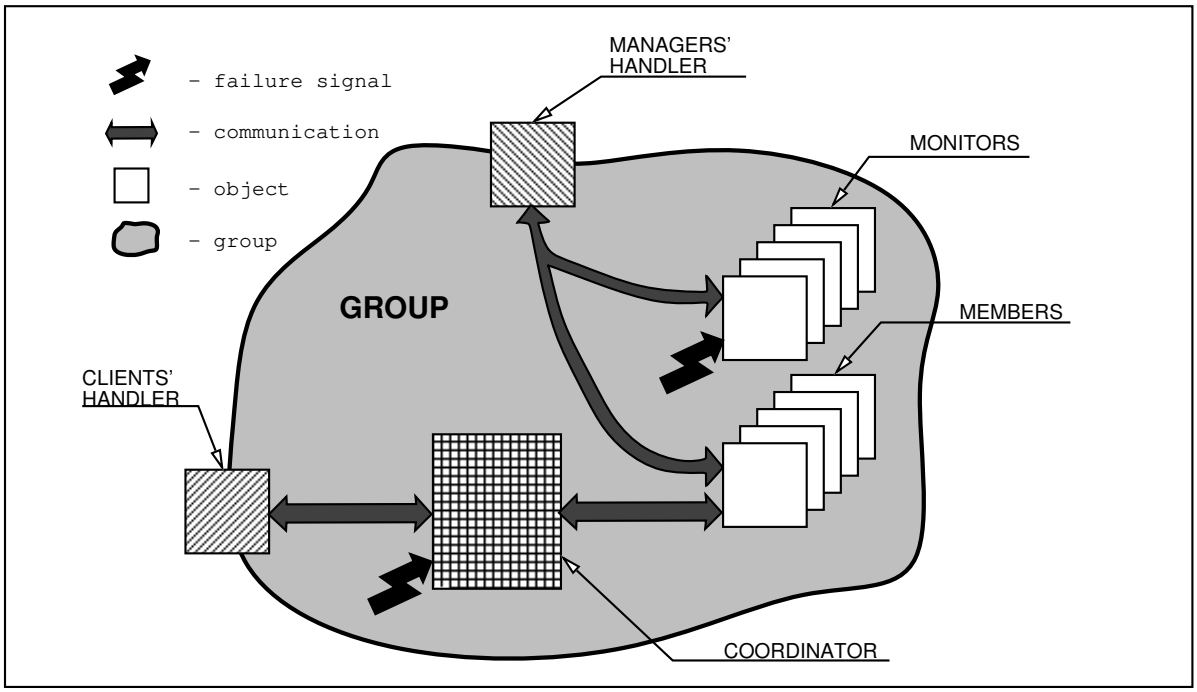

Figure 8: Structure of a group in Emerald

ject. This object conforms to the managerial view of a group. To obtain the client's view of a group, the *GetGroup* method has to be invoked on the group object. The such obtained object has the service type of a group and for a client is indistinguishable from a normal object. This object fully encapsulates all group mechanisms and may be freely used wherever a normal object of the same type could be used.

Three operations related to membership maintenance are provided. Objects may be added to a group, removed from it, membership may be inspected. We treat objects that monitor a group as a special case of member objects. Methods for maintaining monitor objects membership are available.

5.3 Communication protocols

There are three kinds of operations in our Emerald group mechanisms that require communication protocols. They are: membership changes, monitors' membership changes, and group invocations. A protocol to be used in case of group invocations is defined for each operation separately. Two extra protocols are used for maintenance: one for operations that change the state of a group (adding and removal of members or monitors) and one for inspecting the state. We treat members and monitors uniformly so they share communication protocols.

The number and kind of supported protocols depends only on the underlying system. We use the fol-

that should handle different steps of group communication. In this way, a group may be constructed out of reusable parts.

5 Groups in Emerald

Emerald [11] is an object-based programming language and a distributed operating system. The language is strongly typed and supports distribution. Emerald's objects are mobile and may be invoked in a location independent manner. Objects may be active; concurrency control is achieved by monitored sections inside objects. The language provides for a number of built-in objects which are implementations of basic types (like *Integer* or *Array*) or abstract some system level features (e.g., a *Node* object representing an Emerald system kernel).

The core of the Emerald implementation of group mechanism are group objects created by a new built-in object *Group*. They enable creation, maintenance, and monitoring of a group. Coordinator objects encapsulate group behaviour and state. Group communication is possible in them by means of **invocation** operations. Syntax of the Emerald language have been enhanced by the **by** clause that enables a control of the invocation on the callee's side. All the changes to the language do not change the Emerald's object model. Group communication is type secure and supports mobility and distribution.

5.1 Group structure

Figure 8 shows a structure of a group in Emerald. First of all, a group contains a set of members. All members provide the same operations. There exist also secondary members of a group called monitors. Monitors serve for monitoring group's events like changes to its membership or failures. The central object of a group is its coordinator (Figure 10. A coordinator defines actions undertaken upon group invocation and handles group communication. Compared to the scheme from the Figure 7, the coordinator combines functionality of the *Start*, *Init*, and *Reply* objects. It encapsulates also the *Channel* objects whose functionality is invisible. There are two handles to a group that correspond to different views of a group. A clients' handle, corresponding to the client view of a group, provides access to services offered by a group to its clients. The managers' handle, corresponding to the managerial view of a group, provides methods for maintaining a group.

5.2 Group maintenance

A group is created by the *Group* object (Figure 9. The method *Of* on this object returns an object which is a creator of new groups and serves as an abstract type for them. Two types are necessary for creation of a group: a type of member objects and a type of a client's view of the group. These two types are passed as arguments to the *Of* method. To create a new group we invoke the *Create* method on the appropriate group creator. The argument of the *Create* method is a controller object of the group. The controller has the same type as the service type of its group. As a result of creation of a group we receive a group ob-

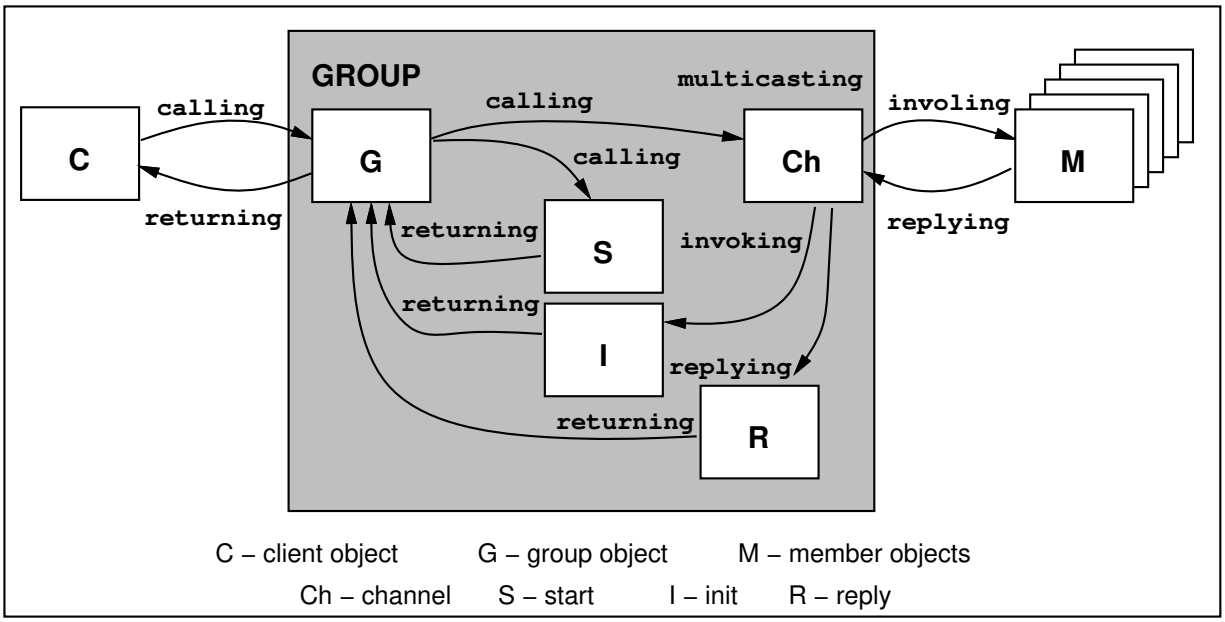

Figure 7: A group – internal view

ordering their invocation relatively to all group invocations in a system. Another group may need ordering relatively to changes in its membership and then selecting.

The results of member invocation are passed back to the *Channel* object which forwards them to the *Reply* object. The *Reply* object encapsulates the policy of collecting results. It is passed each result separately and informed by the *Channel* object when the result is the last expected one. The collecting object may signal the *Channel* object to ignore remaining result. When the final result of an invocation is decided on then it is passed to the *Group* object.

The role of the *Start* and *Init* objects is permitting actions to be undertaken independently of member invocations. There are two events during MOI that mark a change of the state of invocation and unveil more information about its course: starting of the invocation by a group and invoking of members. The two objects allow appropriate reaction to these events, e.g., detection of the case where no members where invoked.

4.4 Properties of the mechanism

The scheme we presented emphasizes each step of MOI and allows to handle knowledge about its course. Client and member object interact with a single object by means of a simple invocation. A group is fully transparent and homogeneous. The mechanism permits creation of new groups by specifying object

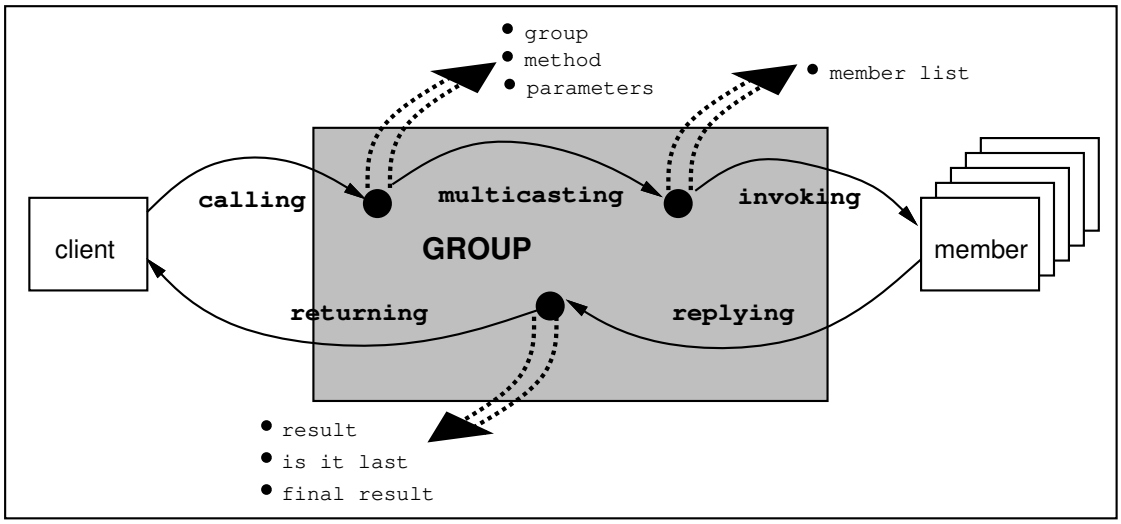

Figure 6: A scenario of MOI

communication should allow communication with an unknown number of unknown objects. If the membership of a group may be changed in parallel to invocations then information about the number and identity of members of a group becomes available at the call time. As this information may be critically important (e.g., algorithms that are based on accurate information about membership of a group) it should be revealed somehow to the receiver of a Multiple Object Invocation.

4.3 A proposed mechanism

In Figure 7, we show a scheme of a flexible object-oriented group mechanism. The mechanism is a framework that allows construction of a group by filling it with appropriate objects. Each object encapsulates a single step of invocation and permits any actions to be undertaken upon such a step.

The *Group* object is a handle to a group and is the only object visible to a client. This object conveys information about how a group invocation should be turned into invocations of members. If a group invocation should cause group invocation of members then the *Group* object invokes *Channel* object and passes to it as arguments the member invocation message and multicast method.

The *Channel* object performs all actions that eventually lead to invocations of members. This includes selecting members to be invoked, ordering of invocation with respect to other invocations, and synchronizing the invocation with other events. We do not split the actions of the *Channel* object into separate steps because some of them may be unnecessary and their order may differ from a group to group. For example, a group may require selecting half of members and

tion message, performing the invocation, and collecting of the results. The only difference between the two types of invocation is that, in the latter, an invocation message triggers execution of the appropriate methods in a number of objects instead of only one object. This leads to forking of new threads of control, one per invoked object. The processes created in the invoked objects proceed asynchronously and this asynchrony may be preserved in handling results: performing invocations and collecting results may overlap. Each result is processed at the moment of its reception without waiting for the others. Additionally, multicast invocation may be abandoned at any moment. If the already collected results are enough then the remaining ones may be ignored.

In Figure 6, we show each step of a MOI. The big rectangle denotes a group object, arcs stand for consecutive steps of MOI, the big arrows point to the lists of information that becomes available to the end of each step. In the first step (*calling*), a client's message is passed to the group. We know at this point a group that is being called, the invoked method, and arguments of the invocation. The next step (*multicasting*) encompasses protocols used for selection of members to be called, ordering of the invocation message relatively to other messages, and synchronization. As a result of this step a list of member objects to be actually invoked is available. The third step (*invoking*) consists of invocations of selected member objects. The next step (*replying*) is returning results by member objects. Knowledge of the list of invoked members allows to detect if the returned result is the last one. The returning the final result forms the last step of execution of MOI (*returning*).

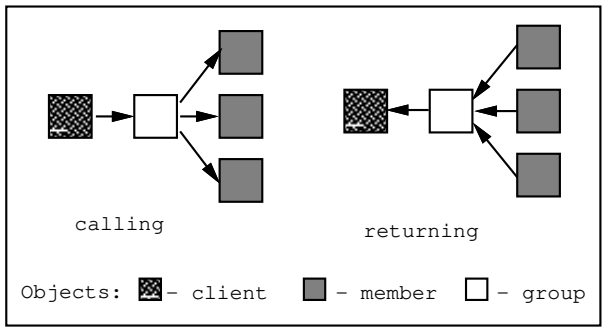

Figure 5: Multiple Object Invocation

A flexible mechanism for group communication should allow to control each step of such an invocation and modify it according to the increasing knowledge. This is particularly important for efficiency—any restriction in possible scenarios of handling group invocation may lead to unnatural and costly solutions. For example, a client may be interested only in consistent results from a majority of group members. He could collect all results and then make a decision about their correctness but given a possibility to abort collecting at any moment he may resume his actions exactly when it becomes feasible, i.e., when a result that allows to qualify existence of majority is received.

Any method on an object may be invoked either directly or by means of the Multiple Object Invocation mechanism. In the traditional invocation all information about it is available at the call time: the identity of a caller and a callee, the method's name and parameters. In contrast, a flexible mechanism of group

group could hide failure handling and recovery mechanisms so that a client sees a clean exception interface.

Different views of a group

A group object plays three roles: it is maintained by some manager objects, it is accessed by its clients, and it interacts with its members. The roles define three views of a group. For its managers, a group is a collection of objects with a number of operations to maintain it. These include creation of such a collection, its dynamic changes, and control over its actions. For a client, a group is a server that offers a certain service. The implementation of such a server, in this case the fact of being a group, may be irrelevant for its client so the client's view of a group does not include managerial operations. For its members, a group is a client. Therefore it is the group that dictates the view of its members because the group expects a certain functionality from its members.

The three presented views have corresponding abstract types.

service type – a type of the service a group is offering to its clients,

member type – a type required of an object to become a member,

managerial type – a type describing operations that allow maintenance of a group.

The service and managerial types are types of a group object while the member type is the required type of the group's members.

An object-oriented group mechanism should permit creation of groups that may be seen according to the three views. It should be possible to separate the views in order to distinguish among different capabilities to a group (service and management).

Homogeneity

According to us, groups should be homogeneous, i.e., all members of a group should have the same functionality. The reason for excluding heterogeneity is an attempt to keep the mechanism simple. We do not loose at the same time any flexibility because heterogeneous groups may be constructed out of homogeneous ones.

Homogeneity of groups is equivalent to the requirement that in a strongly typed object system, there should be one type of members for a group. Even in an untyped language, conceptually, there would be only one expected type of members. The reason is that, in contrast to simple collections (like array or set) that allow objects of all types as their members, a group performs actions on its members and each member has to implement the expected functionality.

4.2 Group communication

Multiple Object Invocation [10] is a method of accomplishing group communication among objects. It is based on the following scheme: an object identifying a group is invoked, this invocation is turned into invocations of the appropriate method on appropriate member objects, results of these invocations are collected and returned as the result of the original invocation.

A Multiple Object Invocation, just like a traditional invocation, consists of three steps: sending an invoca-

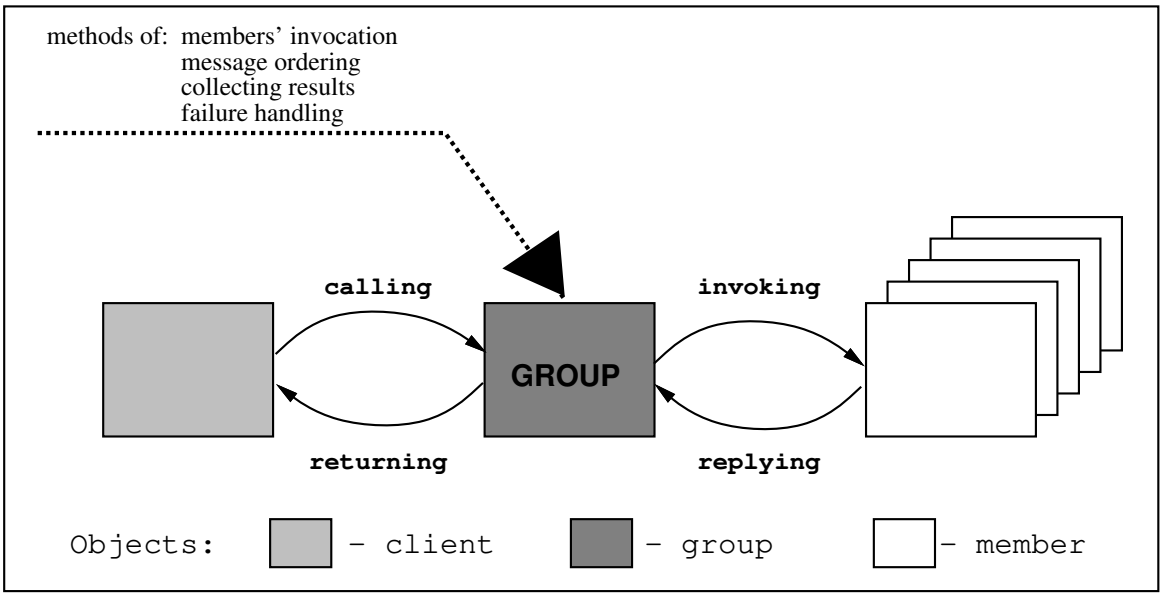

Figure 4: A group – external view

Group as a framework

A group mechanism should be a framework for expressing structure and interactions. A group should encapsulate interconnections of objects and protocols of their communication. It should be a reusable scheme which may be used in composing systems out of ready-made structures. A group mechanism should allow specifying reusable groups by giving a general and flexible frame of a group that could be specialized by defining appropriate components and protocols.

Transparency of group services

An object-oriented group mechanism should be transparent. Neither clients nor members should realize in their operations that they interact with a special kind of object.

Transparency of groups is possible due to duality of a group object (see Figure 4). A group is used in two ways: first, objects may join a group and become its members, second, client objects may access a group asking for some service and these requests may cause the group to invoke its members. A group object is a server for its client and a member object is a server for its group so one can view a group object as an interface between clients and members.

A group mechanism may be transparent if it enables creation of groups that encapsulate all techniques and policies that are used to carry out the group's tasks. In particular, a group has to hide the protocols used for selecting members that are to service a client's request, ordering of messages passed to the group, and collecting of results of members' invocations. Additionally, a

A group may undertake some actions as a response to clients' requests or it may initiate its own independent actions. A group may be monitored, i.e., changes to its configuration or failures may be reported to appropriate objects. A group may also perform some actions to preserve its state or consistency.

Failures of a group include crashes of its members, failures of their actions, and failures of the group's actions. Reactions to failures may realize a number of models. For example, group may detect and mask failures, possibly recreating faulty parts or rerunning failing procedures. A group may also reduce its functionality or performance as a reaction to a failure or declare its crash and stop further actions.

3.3 Intragroup communication

Communication is used within a group in order to exchange information among its parts, to synchronize them, and for maintenance of the group. Definition of communication for a group gives information about which objects communicate with each other and what protocols are used for this communication. Communication may be based on message passing or on some other form of information exchange.

Specification of a communication protocol requires defining the method used (point-to-point or group communication) and the required level of synchrony and fault tolerance. A protocol is reliable if it assures that each operational addressee eventually receives a message passed to it. A protocol is said to be atomic if it guarantees that a message is delivered either to all destinations or to none of them. A protocol may deliver messages without ordering or according to a selected order. Causal order orders only messages that depend one on another and leaves unrelated messages unordered. Atomic order assures that each receiver sees exactly the same sequence of messages (it may be incompatible with the causal order). Atomic causal order delivers messages to all receivers in the same order and preserves causality among them.

4 Groups of objects

In this section, we show how to create groups of objects according to our general model. First, we present the expected properties of an object-oriented group mechanism. We show then how group communication may be achieved among objects. Finally, we propose a mechanism that fulfills the identified requirements. The presented mechanism is designed for a concurrent and distributed system but it is equally applicable for a sequential system or language (some minor issues become irrelevant).

4.1 Requirements for OO-Groups

An object-oriented group mechanism should conform to the premises of object-orientedness: encapsulation, abstraction, and reusability. It should be expressed in terms of objects: a group should be represented by an object and group communication should take the form of object invocation.

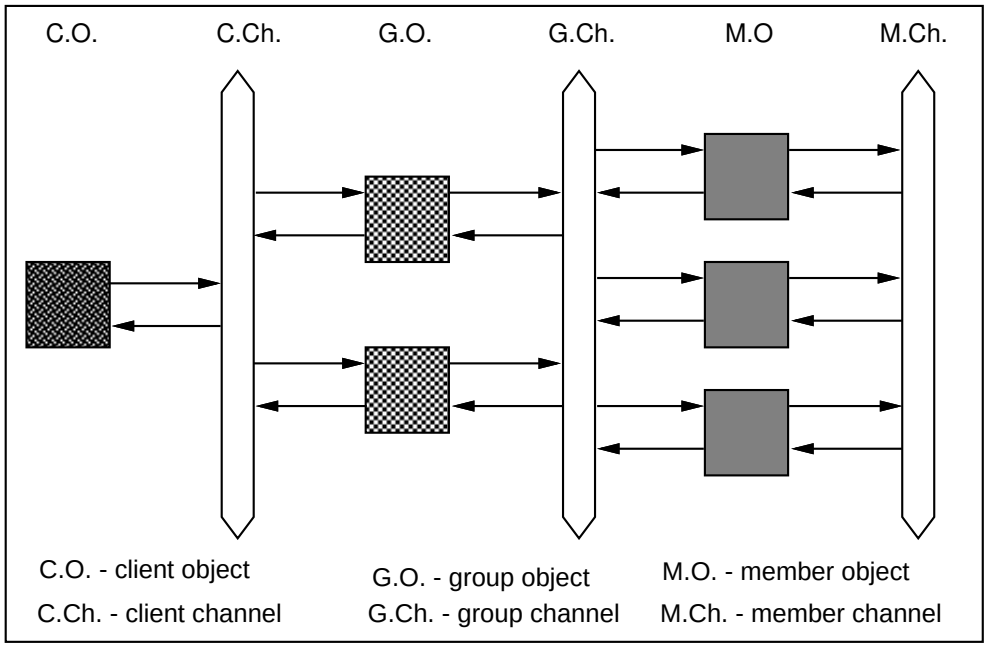

Figure 3: General model of a group structure

face between client objects and a group. Their main role is accepting client's requests, forwarding them to the members of the group, and supervising realization of the requests. **Member objects** are the objects whose cooperation leads to accomplishing the group's tasks. A client's request for a service, passed to a group object, is turned into appropriate actions of member objects. **Channels** encapsulate communication mechanisms used among objects comprising a group. The model permits some groups to have a simplified structure, e.g., member objects may be identified with group objects which allows clients to access them directly.

3.2 Specification of a group

Specification of a group consists of definition of its configuration, its actions, communication protocols used in it, and methods of maintenance and failure handling.

Configuration of a group is defined by a specification of objects that are included in it and their interconnection. A group may be either static or dynamic, depending on whether its membership changes or not. A group is open if the configuration permits clients to access member objects and is closed otherwise. A group may be built according to the client-server model or some other model, e.g., it may be a self-contained peer group that does not allow any external access.

2.4 Candidates for grouping

There is a number of simple guidelines that help in deciding whether a set of objects may be clustered into a group and what kind of application may potentially benefit from using the group paradigm. First of all, if there is an interaction that involves more than two objects then we can apply group communication to simplify it. If a number of objects collectively accomplishes some service then they may be closed in a group. If a set of objects competes for a resource, requires synchronization, or responds to the same events then conceptually they are a group and could be used as such. Below we show some examples of grouping.

parallel computations – a pool of workers, a model often used in parallel computing, may be efficiently implemented as a group. In this case, the group breaks clients' requests into separate tasks, assigns the tasks to workers, and supervises their completion.

Graphical User Interface – if a widget is to control a number of parts of the user interface then the parts may be grouped and assigned to the widget. Interestingly, the widget does not have to be specially programmed for such a group as it is the group that dispatches events.

simulation – groups may help in modeling dependencies among sets of entities that react to the same stimuli. For example, pressing a break pedal causes all attached wheels to block; the pressing person does not have to know how mane wheels are attached and where they are located.

knowledge representation – also modeling of knowledge may include group dependencies, for instance a failure of a module in a machine may generate failures in a number of other modules.

distributed systems – name servers, transactions for replicated databases, and load balancing are among the most spectacular examples of distributed application that may benefit from application of groups.

3 A model of a group

In [10] we proposed a general model of a group. It may be used for expressing the structure of a group, its use, as well as interactions among its parts. The model is general as it tries to grasp the most relevant aspects of design and use of a group without precluding any technique or application. It is also not limited to the object-oriented paradigm; the word object denotes in this section any entity that exists in the language or system used.

In this section, we will present the model and discuss how a group may be specified and used according to it. In the following sections we will apply the model for construction of object-oriented groups.

3.1 Structure of a group

According to our model, a group consists of objects that interface it with the outside world, objects that perform the group's tasks, and links among them (see Figure 3). **Group objects** identify a group and provide access to its services. They are the only inter-

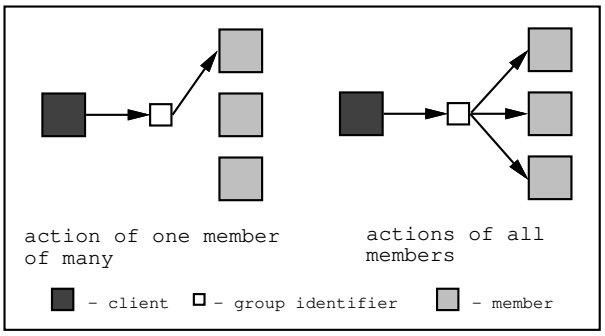

Figure 2: Examples of plurality hiding

tion of plurality[1], allows a group to hide any policy of passing a client's requests to the group's members (see Figure 2). For example, a single-member group may hide identity of an object: clients interact with a group which interfaces them with its member and may change the member transparently into another. In case a group consists of a number of objects, it may choose to pass client's requests to one member or to any subset of the group members.

A group may shield its clients from the its protocols so it may be used to hide complex communication patterns. Group communication simplifies specification of protocols involving sets of objects because it permits to synchronize or pass information to any number of entities in a single action. Use of groups and group communication may therefore simplify interactions inside a system and provide its clients with a simpler interface.

[1] A term coined by Black and Immel in [2]

2.2 Support for replication

Replication is a technique used in distributed systems for achieving higher availability and increasing fault tolerance. These goals are achieved by having a number of servers duplicated in different nodes of a network.

A replicated service may be implemented as a group. Each replica is an independent server and has the same functionality. Clients' requests are passed to the group which changes them into actions of appropriate replicas. Group communication is used for passing information to all servers and preserving consistency among them. A group mechanism gives a uniform method of creating such services and their maintenance.

2.3 Fault tolerant distributed computing

The greatest problem in construction of a reliable distributed system is impossibility of achieving consensus among parts of such a system [5]. Each application that needs fault tolerance inevitably has to use some complex protocols designed to provide some partial solution to this problem. It is argued that fault tolerance may be based on group communication [1] so many of the protocols use this form of message passing. Using a group mechanism together with group communication may thus simplify specification of a fault-tolerant system due to encapsulation of protocols for achieving the required level of reliability.

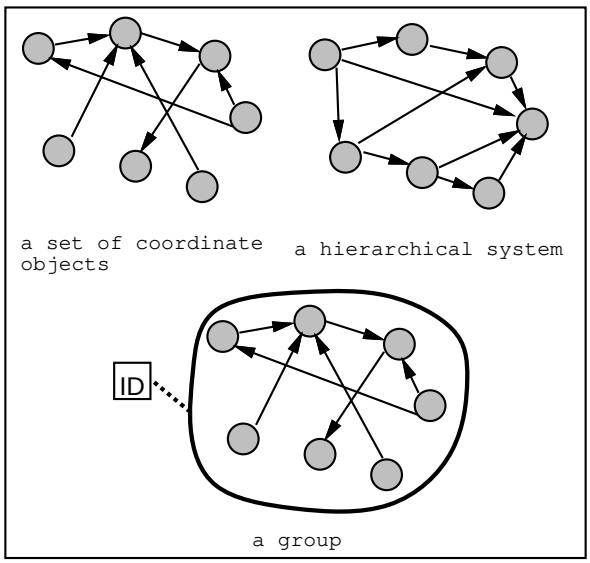

Figure 1: Group

involve an arbitrary number of objects (possibly more than two). It is based on passing information to or receiving it from a number of objects (many-to-many, one-to-many, many-to-one) as opposed to simple communication (one-to-one) in which only two objects interact.

The paradigm of a group is a framework which allows to cluster objects according to patterns of their interactions. It gives a method of specifying a structure of such a cluster, interconnections among its components, and protocols of communication among them. Application of groups simplifies design of a system by allowing to split two aspects: design of the system's components and combining them into a separately designed framework—groups.

In this paper, we present a group mechanism for object-oriented languages. We start with a discussion of applicability of groups and with examples of their use. We propose a general model of a group structure and show important aspects of construction of groups according to the model. We discuss what specific requirements are posed to a group mechanism by object-oriented programming. We describe a mechanism of object-oriented groups and group communication that we created using our model. We present a realization of the mechanism in the Emerald object-based language. We close our presentation by comparing the proposed mechanisms to other approaches to programming with groups.

2 Applicability of groups

Groups and group communication are not a panacea for all troubles caused by complexity but there is a class of problems that may be simplified by application of group mechanisms. These problems are characterized by complex patterns of interactions involving sets of objects. In this section we present a number of such problems and on their example show helpfulness of the group paradigm.

2.1 Hiding complexity

The use of complicated systems does not have to reflect their complexity. A group mechanism may provide a simple interface that hides complexity behind a well defined set of access methods.

A group is seen by its clients as a single entity but it may consist of any number of objects. This hiding of the number of entities, called encapsula-

Group Mechanisms for an Object-oriented Language*

a research paper submitted to OOPSLA'93

Przemysław Pardyak
Institute of Computer Science
University of Mining and Metallurgy, Cracow, Poland
e-mail: *pardy@tulip.ics.agh.edu.pl*

Abstract

This article presents the notion of a group as a valid paradigm for object-oriented programming. A group mechanism provides a framework for clustering cooperating objects. It encapsulates structure of such a cluster and patterns of interactions among its components. This paper is based on the presented general model of a group structure. It shows specification of a group and the use of group communication in the model. It discusses requirements for a group mechanism for object-oriented programming and shows how the model is applied in enhancements to the Emerald object-based language. We discuss applicability of the group paradigm and show examples of its use. We believe that object-oriented programming can profit from the addition of a group mechanism because its application simplifies specification and implementation of complex systems.

*This research was partially supported by Komitet Badań Naukowych under grant no. 8.0077.91.01

1 Introduction

Complex systems may consist of a large number of objects that cooperate in accomplishing their common tasks. Such systems often have elaborate structure and use complicated communication protocols for information exchange and synchronization. Their design and implementation is often difficult because of the lack of an adequate framework for expressing structure and communication.

We propose a group mechanism as a solution to some of the problems caused by complexity. A **group** is a set of coordinate cooperating objects (Figure 1). A group is identifiable and has an access method associated with it. A group has no hierarchy which would imply that some members of a group are subordinate to others. Access method and identity allow to define crisp boundaries of a group. Cooperation requires communication; in case of groups this may take the form of group communication. **Group communication** is an act of information exchange that may