Organising Computation through Dynamic Grouping

Michael Fisher[1], Chiara Ghidini[2,⋆], and Benjamin Hirsch[1]

[1] Department of Computer Science, University of Liverpool, Liverpool L69 7ZF, UK
{M.Fisher,B.Hirsch}@csc.liv.ac.uk
[2] Automated Reasoning Systems Division (SRA), ITC-IRST, Trento, Italy
ghidini@itc.it

Abstract. There are a range of abstractions used for both modelling and programming of modern computational systems. While these abstractions may have been devised for specific purposes, the variety of options is often confusing, with development and programming techniques often being distinct. The aim of this chapter is two-fold. First, we bring together a number of these abstractions into one, general, view. Second, we show how, by grouping computational elements, this general view can capture a range of behaviours in areas such as multi-agent systems, web services, and object-oriented systems. This framework then provides a basis for design and implementation techniques for a wide variety of modern computational systems, in particular providing the basis of a general programming language for dynamic, distributed computation.

1 Introduction

In the past there were just *programs* and *procedures*. Then *processes* and *objects*. Now, we also have *components*, *agents*, *software features* and *web services*. While these abstractions are used in a variety of different application areas, there is a great deal of similarity between these concepts. All concern independent, typically concurrent, entities whose basic dynamic behaviour is modified by the specific contexts in which they are used. Thus,

- the behaviour of an object-oriented component is modified by its position within an object hierarchy, typically its relationship to other components represented via inheritance,
- the behaviour of an agent is modified by its own internal goals, together with its views of other agents,
- the behaviour of a feature is modified by the other features present, as each feature only represents a partial view of the computational element, while
- the behaviour of a web service is modified by the ontology it uses, together with the web context within which it is invoked.

And so on.

The range of abstractions used in both modelling and programming is often confusing. When designing modern software, are we concerned with objects or agents? Should

⋆ Work supported by MIUR under FIRB-RBNE0195K5 contract.

M.D. Ryan et al. (Eds.): Objects, Agents, and Features, LNCS 2975, pp. 117–136, 2004.

these distributed elements be seen as services or agents? Should they be components or objects? And so on. In an effort to simplify this situation, we here bring together a number of these abstractions into one, general, view. Later, we will show how, by grouping computational elements, this general view can capture a range of behaviours in the above areas. This framework then provides a basis for design and implementation techniques for a wide variety of modern computational systems, in particular providing the basis of a general programming language for dynamic, and potentially distributed, computation.

1.1 Concepts

While there is often little agreement on the definitions of the elements we consider in this chapter, we will use the following working descriptions.

Object: An object is a computational entity encapsulating both data and behaviour [15]. Objects react to messages received by invoking *methods*. In object-based systems where inheritance is present, messages that can not be dealt with by an object may be passed on to other objects using inheritance. Often, but not always [12], objects are clustered into *classes* in order to capture common functionality/behaviour [7].

Agent: An agent is an *autonomous* computational entity encapsulating data and behaviour [17]. In contrast to objects, agents have control over how they react to their environment, and may adapt their internal behaviour to evolving situations [6]. Agents often work in unpredictable environments and so may have only subjective representations of aspects outside their control [16].

Software Component: A component is a software object encapsulating specific functionality and interacting with other components through its defined interface. A component has a clearly defined interface and conforms to a prescribed behaviour common to all components within the software architecture [11, 9].

Feature: A feature is an optional unit of functionality that may be added to, or excluded from, a system [4]. A feature is usually perceived by the user as having a self-contained functional role. Features may override the default behaviour of the system, or introduce a new behaviour. As a consequence they may interact or interfere in unexpected ways and thus cause the overall system behaviour to be undesirable [10].

Web Service: A software system identified by a URI, whose public interfaces and bindings are defined, and described, using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by its definition, using XML based messages conveyed by Internet protocols [13]. This framework provides the basis for ambitious projects such as the Semantic Web [1, 14].

1.2 Grouping

In this chapter we will primarily consider the *organisation* of computation within *collections* of these entities. In particular, we show how a simple framework of computation within groups can provide a basis for a range of activities of contemporary software artifacts such as those above. Thus, grouping together each of these types of artifact give us a range of different possible behaviours for each type:

Objects. The grouping of objects together is widespread in class-based systems as classes can be seen as groups of objects sharing common properties. Similarly, inheritance can be seen as the linking of groups of objects. More interestingly, grouping can be seen as a practical mechanism for both structuring the object space and implementing forms of communication and fault-tolerance [3, 2].

Agents. It is common, within agent-based systems, to deal with large, dynamic, groups of agents. In such multi-agent systems, sub-grouping is not only used to model both cooperative and competitive structures, but can also be used to organise the agents into structures corresponding to physical situations (e.g. all agents in the same vicinity are grouped together). In all these cases, it is natural to have the notion of dynamic movement of agents between groups and the dynamic creation and destruction of groups [17].

Components. In component-based software engineering [11], it is the construction of the group of components, rather than the construction of single components, that is important. A collection of components forms a *component architecture*, and each component architecture defines the conventions by which components are linked together. Collecting together suitable components and ensuring they interact in an appropriate way is the key mechanism, though such collection is usually carried out by the programmer/user [9].

Features. A increasingly common way of designing complex systems consists in thinking of the system as being composed of a basic part plus a collection of features that can be incrementally added to the system to provide the required services. In such systems, it is the appropriate integration of features that is crucial, as features may interact with each other in ways that are difficult to anticipate, and may cause the overall system behaviour to be undesirable [10].

Services. The idea behind web services is that appropriate services can again be collected together in order to provide an application working in a distributed way over the WWW. The difference between this approach and that of many (but not all) of the others above is that the services are typically meant to be found and grouped automatically. This search for appropriate services is supported by the the idea, from the *Semantic Web*, of publishing a service's properties and then letting other agents/applets browse/select these published specifications in order to choose the most appropriate service [14, 1].

1.3 Structure of the Chapter

The structure of this chapter is as follows. Our general approach to the organisation of entities in collections (or groups) is outlined in Section 2, where we present some simple examples of activities involving collections of entities. Our goal is to provide a framework where computation is organised within collections of entities provided by groups. Thus, Section 3 presents the implementation of the grouping mechanism presented in Section 2 together with further examples. Section 4 contains higher-level views of how to use our framework in a variety of application areas and, finally, Section 5 provides some concluding remarks.

2 Elements and Groups

In order to avoid calling the basic elements we deal with by some name that already has numerous definitions, e.g. 'objects' or 'agents', we will simply call them 'elements'.

2.1 Elements

Elements are the basic entities within our model. Elements:

- are opaque, encapsulating both data and behaviour;
- are concurrently active, in particular they are asynchronously executing; and
- communicate via message-passing.

Specifically, the basic communication mechanism we provide between elements is *broadcast* message-passing. Thus, when an element sends a message it does not necessarily send it to a specified *destination*, it merely sends it to its environment, where it can be received by *all* other elements within the environment. Although broadcast is the basic mechanism, both multi-cast and point-to-point message-passing can be implemented on top of this.

The default behaviour for a message is that if it is broadcast, then it will *eventually* be received at all possible receivers. Note that, by default, the order of messages is not preserved, though such a constraint can be added if necessary. Individual elements only act upon certain identified messages. Thus, an element is able to filter out messages that it wishes to recognise, ignoring all others.

2.2 Groups

A group contains a set of elements. Groups understand certain messages, in particular those relating to group manipulation, such as adding, deleting, etc. Thus, the basic properties we require of groups are that elements should be able to

- send a message within a group,
- ascertain whether a certain element is a member of a group,
- add an element to a group,
- remove a specified element from a group, and,
- construct a new subgroup.

There are other properties that might be useful, such as the ability to list the members of a group, but they are not essential.

Groups contain (references to) other elements which may, in turn, be either simple elements or groups. Groups may also overlap. Thus, one element may be a member of several groups. When an element within a group sends a message, the default behaviour is to pass the message on to all the members of the groups.

While the basic purpose of groups is to restrict the set of receivers for broadcast messages, and thus provide finer structure within the element space, more complex applications of groups can be envisaged. For example, the group might enforce a different internal model of communication [5]. Thus, groups can effectively enforce their own 'meta-level' constraints on message-passing between elements, for example specifying that the order of messages is preserved.

2.3 Element ≡ Group

To simplify and clarify the model, we actually identify the notions of element and group. Thus, groups are also elements. More surprisingly, perhaps, elements are also groups. Consequently, basic elements are groups with no contents. One of the values of identifying these two concepts is that any message that can be sent to an element, can be sent to a group (concerning cloning, termination, activation, etc) and vice versa (concerning group membership, addition to groups, etc). Thus, not only can elements be members of one or more groups, but they can also serve as groups for other elements. Further advantages of this approach are:

- communication (which, in our approach, is based on broadcasting) can be limited to particular networks of groups;
- since elements have behaviour, and groups are also elements, groups can also have internal policies and rules. Therefore groups are much more powerful than mere "containers"[1];
- the inherent structure can be exploited to group elements based on different aspects such as problem structure, abilities, and even meta-information such as owner, physical location etc; and
- groups can capture key computational information, e.g. groups can be transient, can be passed between elements, and can evolve, while long-lived groups may also capture the idea of "patterns of activity".

Thus, from the different behaviours of elements, different group behaviours can follow. And vice versa.

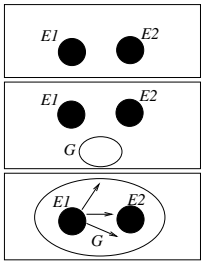
Finally, it is important to note that, while groups are initially designed/programmed with a specific behaviour (e.g. controlling entry to the group, controlling communication within/into the group, or controlling how information is shared within the group), in principle they can 'learn'/adopt new behaviour as computation proceeds. Thus, the group of elements can evolve, both in terms of content and in terms of behaviour, over time.

[1] As an example, in [8] we showed how simple rules could be used to formulate different behaviours within agents, and how those behaviours influence the resulting group structures.

2.4 Simple Examples

We consider here three simple, yet very common, examples of behaviour involving elements belonging to some sort of collections, and we illustrate how to represent these behaviours in our framework.

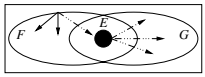
Example 1 (Point-to-Point Communication)
In order to provide a mechanism for element E1 to communicate directly to (and only with) element E2, E1 may



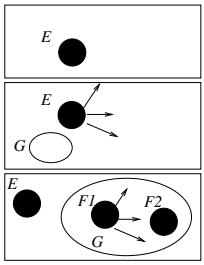
1. *make a new element, called G, and add itself to G,*
2. *broadcast a message suggesting E2 joins G, to G,*
3. *then broadcast a message, m, within G.*

Note that the decision about whether E2 actually joins G is likely to be up to E2 itself. Similarly, we are not considering security aspects here, for example ensuring no other elements, for example E3, join G.

Example 2 (Filtering)
Element E is a member of both F and G. When F broadcasts messages to its contents, E will receive these and can pass on selected messages to G. Thus, E acts as an intermediary between F and G.

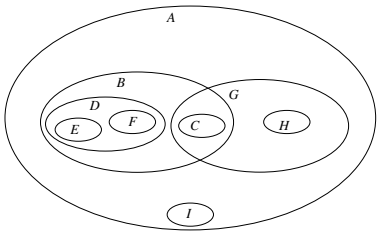


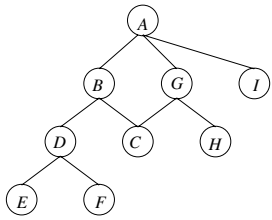
Example 3 (Interest Groups)
Element E wishes to collect together all those elements that have interest in, or capability to solve, a certain problem. Thus, E creates a new element, G. E then broadcasts a message, with G as one of its arguments, asking for interested parties to join G. (Note that the message has sufficient information encoded within it for any receiving element to be able to work out what capabilities are being requested.) When such an element, say F, receives this message, it can join G. Thus, G contains all those elements who have joined based on this message and messages broadcast throughout G will just be received by those elements.



3 Implementation

We have implemented the general grouping approach, as outlined above. In order to provide the basis for as wide a range of applications as possible, this implementation is based on Java. Thus, elements are essentially Java threads, with additional classes for organising group structures and communication.





Element	CN	CX
A	{B,G}	∅
B	{D,C}	{A}
C	∅	{B,G}
D	{E,F}	{B}
E	∅	{D}
F	∅	{D}
G	{A}	{C,H}
H	∅	{G}
I	∅	{A}

Fig. 1. Different views of the system structure

3.1 Implementing Groups

As mentioned before, we identify the notions of elements and groups. So, every element has the potential to contain other elements. What we consider to be "basic elements" are essentially elements with empty contents. This organisation is reflected in each element by using three sets: *Content*, *Context* and *Known*. Thus, each element has

- a *Content* set, describing the set of elements that are *members* of the element (group),
- a *Context* set, describing the set of groups (elements) that the element is a *member of*, and
- a *Known* set, describing the set of *known* elements, typically outside the content/-context hierarchy.

Fig. 1 illustrates a typical group structure of our system, the corresponding hierarchical structure, and the representation of this structure using the *Content*/*Context* (**CN**/**CX**) mechanism. Note that we will use different representations throughout the chapter, as they are more or less suited to different situations.

While the pure *Content*/*Context* approach (that is, without the third set *Known*) is simple and intuitive, it has problems. In order for an element to send a message to another, far-away, element, it has to know the topology of the element space in order to avoid broadcasting messages to the whole system. Thus, each element also keeps a list of *known* elements, with which it can communicate directly. When an element encounters a previously unknown element, it adds it to the *Known* set. This allows efficient communication with elements outside the current hierarchy/structure. It should be stressed at this point that we include the *Known* set for convenience and efficiency, rather than pressing theoretical matters. We chose to implement a third set of direct connections as we aim to build a practical tool, and wish to reduce the broadcasting of messages to the entire system when elements have to communicate with other, far-away, elements. Further, it may be difficult to see how such a *Known* set may be established in the first place. With regard to this there are two points to note:

1. since the group structures are dynamic, elements which reside in a common *Content*/*Context* hierarchy at one point may not be so at a later moment in time — thus, retaining references to relevant elements may be useful;

2. even if two elements are in the same *Content*/*Context* hierarchy, it may be more efficient for one to store a direct reference to to the other than for general communication to be used.

Since the topology of the element space is dynamic, it can adapt to accommodate new structures to solve problems. This is typically achieved by two general methods. First, an element can move through the element space and join a group that can help it solve its problem (or alternatively it can invite other elements to join its content). Note that "movement" does not necessarily mean that elements leave one group to join the other — they can join many groups and have many elements in their content. Also, movement refers to virtual, rather than physical, movement. Due to the ability to send messages to sub-groups, elements can restrict the amount of superfluous messages further. Alternatively, elements can create "dedicated" group elements that serve as containers for elements that share some property (typically, a common ability).

Whether elements are invited to *Content*, *Context*, or to dedicated groups, the duration of their stay can be either transient (that is, they do what they are asked and then leave), or (semi) permanent (that is, elements stay in the group even after they have completed their task). Duration of stay depends on both the element and its context.

3.2 Implementing Communication

As described above, elements are implemented using threads. Attached to each element is a *message buffer*, effectively providing the *inbox* for that element. Thus, messages, which are instances of a specific `Message` class, are added to an element's message buffer in order to implement communication. Typically, elements read all messages received in a given cycle and act upon them. Naturally, they can also send messages and/or carry out computations during a cycle.

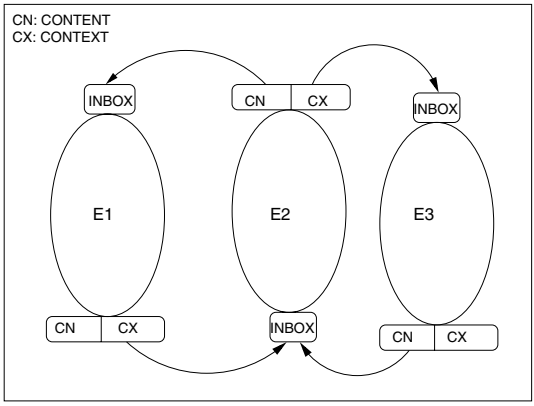


Fig. 2. Implementation of communication

Fig. 2 illustrates how we implement communication by providing a particular example. Each element has an *inbox* to which others can link, either through their *Content*, or

Context (or *Known*, which we do not show in the picture). Generally if an element connects to another one through it's *Content*, there will be a reciprocal connection through the latter's *Context*, and vice-versa, in order to comply with the grouping theory.

While the basic requirement for communication is to be able to send messages to elements in the *Context*, *Content*, or *Known* sets, we often require more sophisticated message-passing behaviour. Thus, instead of sending messages just to *Content*, *Context*, or *Known*, an element can specify messages to be sent to an arbitrary subset of elements from these sets. This is achieved by the statement

```
send(Target,Msg)
```

where `Target` is a target set and `Msg` is the instance of `Message` being sent. Target sets can be specified using the set operations *union* (S_1 `union` S_2), *intersection* (S_1 `intersection` S_2), and *exclusion* (S_1 `without` S_2), where S_1 and S_2 can themselves be single elements, sets of elements, or set expressions[2]. Thus,

```
send(Content without i, message)
```

sends `message` to all members of the element's *Content* set *except* element `i`.

Additionally, messages can be nested. Thus, if an element receives a `send` message, it will interpret the `target` relative to its own *Content*/*Context*/*Known* sets. For example, element `i` can send the following message

```
send(Context, send(Content without i, message))     (1)
```

Then, all elements in `i`'s *Context* receive the message

```
send(Content without i,message)
```

and re-send `message` to all their *Content*, except for element `i` (which originally sent the message). So, using (1), `i` can send messages to elements that are "on the same level" as itself, i.e., that share at least one of the same *Context* elements.

Since nesting of message sends is so common, we also have a special send command `sendAll`, which is interpreted by elements to recursively re-send the message to the same set (relative to the receiving element). For example,

```
sendAll(Content, m)
```

will ensure that message `m` will be recursively sent to all members of *Content* sets 'below' the sending element. Also,

```
sendAll(Content union Context, m)
```

will ensure that `m` is propagated to the whole element space. The implementation of `sendAll` is based on the equivalence (when expanded within element `i`)

$$\texttt{sendAll}_{\texttt{i}}\texttt{(Set,Msg)} \equiv \texttt{send(Set,Msg)} \wedge \texttt{send(Set,sendAll(Set}\setminus\texttt{i,Msg))} \quad (2)$$

[2] Note that we sometimes use the mathematical notations $\cup$,$\cap$, and $\setminus$ for the set operations `union`, `intersection`, and `without` respectively. We also drop subscripts whenever possible to enhance readability.

For example, element `i`, upon receiving `sendAll(Content union Context, m)`, will execute two sends:

`send(Content union Context, m)`, and

```
send(Content union Context,
      sendAll((Content union Context)\i, m)).
```

By removing itself from the list of elements that are to receive the message, the element avoids messages being sent over and over again through the element space.

3.3 Primitive Actions

Before listing some of the primitive actions available to an element, we note that the exact implementation of these actions effectively specifies the level of autonomy the agents have. For example, if the elements have a high level of autonomy, then messages sent to them involving movement are just 'suggestions' — the element still decides itself whether to move. However, if we require little autonomy (e.g. typical within object-based systems), these primitives might be implemented to succeed automatically. And there can be a range of different levels between these two extremes. Thus, in describing the primitive and composite actions in this, and the next, section, we must bear in mind the effect that the level of autonomy required may have.
We begin with simple primitives.

`send(Set,Message)` and `receive` are arguably the most important primitive actions. As stated above, messages are sent to a subset of the union of *Content*, *Context*, and *Known* sets. `send` understands not only basic, but also composite sets, such as those described in Section 3.2. `receive` reads *all* messages that are in the element's message buffer and processes them.

`sendAll(Set,Message)` works in a similar way, with the difference that the message is sent twice, once as normal message, and once wrapped into another `sendAll` message. This message is then interpreted by the receiving elements in order to re-send the original message to their respective sets. Each element adds itself to the exclusion list (as in (2) above) so as to avoid receiving the same `sendAll` message again. This method, while very intuitive, does not completely avoid messages being sent more than once to an element, as the message may be able to reach it through several different routes.

`addToContent(Element)` and `addToContext(Element)` are the primitives that allow elements to move within the element space. The element receiving such a message adds the argument to the appropriate set and replies directly back to the argument element with an `addedToContent` or `addedToContext` message. Upon receiving confirmation, the initiating element moves the message buffer of `Element` from *Known* to the appropriate set. This method reduces messages sent though the element space.

`disconnect` is sent if elements want to remove an element from *Content*, *Context*, or *Known*. If received, the element replies with `disconnected` and removes the relevant

connection. While elements can't refuse to be disconnected, we adopt the request/acknowledge technique in order to give the element that is to be disconnected a chance to send final messages before severing the connection.

`create` allows an element to create a new element. This is mainly intended for creating groups in order to collect elements together[3].

3.4 Composite Actions

With the above actions we can now define several higher level behaviours. Again, they primarily concern the movement within the element space.

`moveUp(Set)/goUp(Set)` allows an element to move up within the hierarchical structure (into the group(s) of its context elements). While `moveUp` is initiated by the element that moves, `goUp` would be sent to an element by one of the elements in its *Context*. Fig. 3 illustrates the difference between the two composite actions. The figures on the top represent the group structure before the composite actions are executed, while the bottom part represents the new group structures after the execution of the composite actions.

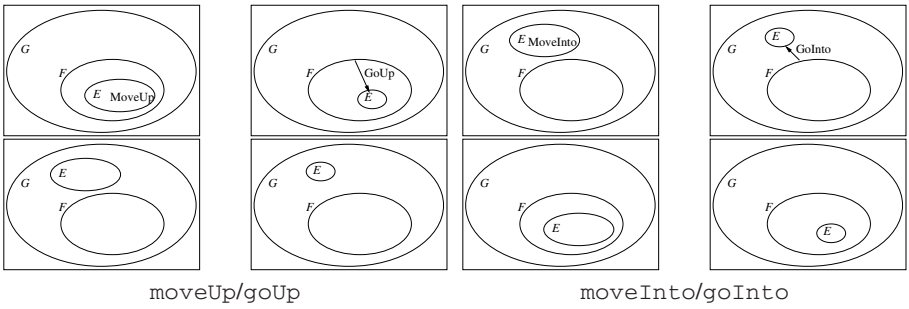


Fig. 3. The composite actions `moveUp`/`goUp` and `moveInto`/`goInto`

`moveInto(Set)/goInto(Set)` is the reverse of the up-movement, as can be seen in Fig. 3. An element moves into one (or more) of its *Content* elements. Again, `moveInto` is initiated by the moving element, while `goInto` is initiated by the one that will receive the moving element.

The `die` predicate does what it says — it makes an element stop its computation gracefully. After receiving such a message, the element sends `addToContent` or `addToContext` messages for each element in the respective sets to *Content* respectively *Context*. Elements in *Context* can add the content elements to their content and

[3] At present, only hard-coded elements can be created, but there is infrastructure in place to allow elements to determine what messages the created element should understand, and how to react to them.

vice versa. It then sends *disconnect* messages to all elements, and stops the thread[4]. Note that in order to just remove an element, one can simply send a disconnect message to *Content* and *Context*, and stop the thread.

`merge(Element)` allows elements to inherit their *Content* and *Context*, such that all *Context* elements of the one being assimilated will be in the *Context* of the assimilating elements, and ditto with *Content*. (Note that there are a number of issues concerning this operation that we will not consider here, for example how to combine the behaviours of elements as well as the contents.)

`clone` makes a copy of an element. As an element is not only defined by its internal programming but, to a substantial part, by its connections with other elements, a "real" clone needs to negotiate connections with those. (Note that we cannot guarantee a clone operation will succeed, as elements might refuse to allow cloned connections.) While the clone operation does attempt to re-connect, we must recognise that this might not always be possible.

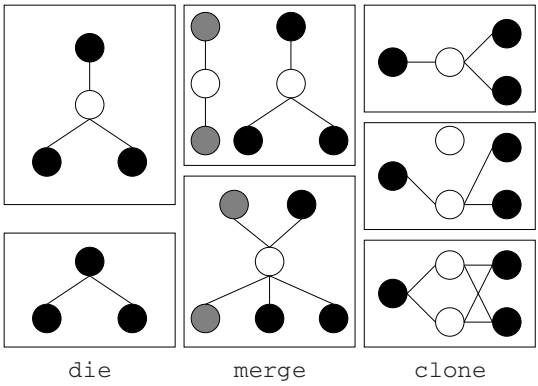


Fig. 4. The composite actions `die`, `merge`, and `clone`

Fig. 4 gives a graphical representation of the effect of the composite actions `die`, `merge`, and `clone` on the structure of the system. The composite actions are initiated by the element(s) represented as an empty circle.

3.5 Messages

In our framework, messages are instances of class `Message`. Each message contains a message name (predicate name) and an arbitrary number of arguments. For ease of use, we refer to arguments using strings, rather than their position (as is common).

[4] The reasoning behind connecting the *Content* and *Context* elements is to avoid disconnected (sets of) elements. While it is possible to re-connect using the underlying *requestChannel* method, it would require elements to know at least one element by name. At this stage, we assume that all elements are connected in one graph.

```
//  createP2P takes the name of the group to be
//  created, as well as the element we wish to talk to,
//  as arguments, and returns the new group element

Element createP2P(String elementName,Element e)
{
  Element p2p = new GroupElement(elementName);
  p2p.addToContent(this);
  p2p.addToContent(e);
  new Thread(p2p).start();
  return p2p;
}

//  ... create the group:

Element G = createP2P("G", E2);

//  ... have something to say:

Message m = new Message(this, "send");
m.addArg("message",message);

//  ... have some set to say it to:

m.addArg("set",new SetExpression("content"));

// ... and say it!
send(new SetExpression("element", "G"),m);
```

Fig. 5. Code fragment defining group supporting point-to-point communication

Arguments can be either part of the actual message content, or meta-data such as sender, messageID, timestamp, etc.

3.6 Code for Simple Examples

This section attempts to provide the reader with a flavour of what the actual code for programming elements looks like. We show how to implement the examples outlined in Section 2.4. While these might be very simple examples, they provide an indication of the ease with which we can cater for different requirements. For readability we omit details such as exception handling, as it adds neither to readability nor understanding of the code.

Example 4 (Point-to-Point Communication) *In Fig. 5, a simple mechanism for point-to-point communication is implemented. The method* `createP2P(String, Element)` *creates a new group element (based on the class* `GroupElement` *which basically just forwards messages) containing itself and the element it wishes to communicate with.*

Example 5 (Filtering) *This example, given in Fig. 6, shows the method* `parseMessage(Message)`*, a method which is called for each message that is read at one cycle. The super class provides standard behaviour for the above mentioned basic and composite actions. Generally, at least a subset of those will be adapted by elements. The code fragment shows how elements can act as intermediaries between different groups (elements). Here, the actual decision on whether or not to forward a message is carried out by the method* `isValid(Message)`*. If the message should be forwarded, the element creates a new message* `send(Content,Message)`*, which is sent to group G.*

```
 void parseMessage(Message m)
{
   // check whether m is to be forwarded

   if (isValid(m))
  {
     // yes, so send m to content of receiving group
     Message new_message = new Message(this, "send");
     new_message.addArg("set",
                        new SetExpression("content"));
     new_message.addArg("message", m);
     send(new SetExpression("element","G"),new_message);
  }
   else   // if not, we do whatever we should be doing
     super.parseMessage(m);
}
```

Fig. 6. Code fragment defining message filtering behaviour

Example 6 (Interest Groups) *In this example, presented in Fig. 7, we show how an element creates an interest group (in this case it looks for other elements that are able to add two numbers), and broadcasts an invitation to the element space.*

4 Applications

While we considered detailed implementation of grouping structures in the last section, we here take a broader view of the form of groups we are developing. In particular, we consider, at a high-level, how such grouping might be used in objects, agents, features and services.

4.1 Objects: Sharing Using Classes and Inheritance

As an exercise, we can use the group structuring mechanism to simulate an object-oriented class hierarchy. Generally, object oriented programming languages offer at least some of the following features [3]: encapsulation; inheritance; polymorphism; abstraction.

```
// Create the group element
Element group = new GroupElement("group");
System.out.println(this+" Created group" group);
new Thread(group).start();
System.out.println(this+" Started group element");

// Add new group to creating element's context
addToContext(group);

// Create message representing ability required
Message needed = new Message();
needed.setContent("do");
needed.addArg("ability","sum");
needed.addArg("arg1",new Integer(2));
needed.addArg("arg2",new Integer(3));
Message a = new Message(this,
                        "neededAbility",
                        new NamedList("needed",needed));

// Send message to initialise group element
send(new SetExpression("element",group),a);

// Create message to find element able to
// undertake the requested ability
Message question = new Message("isAble");
question.addArg("ability","sum");
question.addArg("arg1",new Integer(2));
question.addArg("arg2",new Integer(3));
question.addArg("goTo", group);

// Send 'finding' message to all other elements
sendAll(new SetExpression("without",
       new SetExpression("union","content","context"),
       new SetExpression("element",group)),question);
```

Fig. 7. Code fragment creating an interest group

Encapsulation we get for free, as elements by their very nature encapsulate both data and behaviour.

Inheritance can be simulated in a straightforward manner. *Context* and *Content* contains a link to the "super-element" and children respectively. Now, whenever an element encounters a method it does not implement, it sends a request to its *Context*, which in turn either sends back the computed value, or again forwards the query to its super-element. Note also that we natively can support multiple inheritance, if desired, by allowing more than one element to be in the *Context*.

Polymorphism entails that objects can override methods they would otherwise inherit from their super-class. This again is straightforward in our system, as we only need to provide the element with the relevant methods.

Abstraction, i.e. abstract methods, as well as classes, can also easily be integrated. Recall that joining groups works by sending a request to the group element, which then can decide whether or not to allow the asking element to join. Groups can simply only allow elements into their *Content* that adhere to certain rules — in this case they would need to provide a certain method, or a set of methods.

4.2 Objects: Sharing through Groups

In the majority of object-based systems, object behaviour is shared by using *inheritance*. However, the utility of inheritance relies on the assumption that the inherited behaviour is 'good', and is delivered reliably. We believe the first constraint can be seen as violating the basic principle of objects, i.e., that they control their own behaviour[5], while the second does not sit well in either a distributed or concurrent environments where faults may occur.

We can provide an alternative mechanism: if an object can not deal with a message on its own, it broadcasts a request to its *Context* asking how to do something, e.g., implement a particular method. The object collects the replies and *decides for itself* which one to use. The decision/resolution process within the object might be simple, e.g., choose the first reply, or choose the reply sent by the 'most preferred' object, or could be much more complex, e.g. based upon the 'goodness' of the proposed solution! The important fact there is that the object can decide what to do according to the rules that describe their behaviour.

This approach can be seen as a hybrid between objects and agents, with the elements concerned being allowed a small degree of autonomy in choosing the 'best' solution.

4.3 Agents: Cooperation and Competition

By using groups and broadcast message-passing, we are able to represent typical complex interactions among agents, namely cooperation and competition.

Imagine a scenario where agents are competing, for example by *bidding* for resources. In this case, cooperative activity within groups of agents can be organised so that the group as a whole puts together a bid. If successful, the group must (collectively) decide who to distribute the resource to. Thus, a number of groups might be cooperating internally to generate bids, but competing (with other groups) to have their bid accepted.

Within a given group, various subgroups can be formed. For example, if several members of a group are 'unhappy' with another member's behaviour, they might be able to create a new subgroup within the old grouping which excludes the unwanted agent. Note that members of the subgroup can still receive the outer group's communications, while members of the outer one cannot receive the inner group's communications by default (that is unless an explicit message is sent to them by the inner group). Although we have described this as a retributive act, such dynamic restructuring is natural as groups increase in size.

[5] Often in implementations of inheritance, behaviour (typically method bodies) is copied directly into the object from where it is inherited.

Although these examples have been based upon agents competing and cooperating in order to secure a certain resource, many other types of system can be represented. In particular, in defining the appropriate behaviour for the entities in the system we can represent *agent societies*, where elements seen as individuals cooperate with their fellow group members, but where the groups themselves compete for some global resource.

4.4 Feature Interaction

A complex system can be described as some set of basic behaviours that can be extended with extra features. A prominent example is a phone system where features such as "ring back when free" are added. We may use our system to model this as follows. The main group sends all events that can happen (such as placing a call) to its *Content*. In this content, we have elements (intuitively corresponding to individual features) containing rules that describe the actions that need to take place given an event. The basic system just has the base element as *Content* member. We can, however, add feature elements to the *Content*. They also receive the events, and can act accordingly.

While this is straightforward, it does not solve the problems of feature interaction. We can, in principle, utilise techniques that have been (and are being) produced in the field of feature interaction detection and resolution [4]. In particular we may think of applying different approaches to on-line techniques depending on whether the control of the event is located at the group level (thus using the group as a feature manager), or at the individual feature level (thus allowing features to engage in negotiation processes to find an acceptable solution). Finally, an interesting aspect of this view is how, in some cases, the interaction of features (e.g. complementary or conflicting) can be seen as a variety of interaction between agents (e.g. cooperative or competitive).

4.5 Agents: Varieties of Organisation

In the discussion up to now we "told" elements whether or how to create and manipulate groups. As we increase the autonomy of elements and start talking about agents, we can leave the decision about the method of collaboration to the agents themselves. In the following we identify four different prototypical methods that agents could deploy in order to organise themselves.

Using *no groups* is the base case. Agents are structured in a pre-defined graph, and do not employ groups at all — if they need to collaborate / communicate with other agents, they do so solely by sending messages though the space (either by broadcasting, or using knowledge of the pre-configured space and routing messages to other agents). This case is not very interesting for our approach.

Going a step further, agents can *invite into their Content* others with which they want to communicate. This way, communication can be kept to a minimum while enabling agents to harness the power of the concept of identifying agents and groups. The inviting agent has control over which agents can join, as well as what messages to forward between agents in its *Content*. While it might be advantageous to have this tight control, it also means that agents have to have precisely written control structures — it is not possible to define agent wide policies (e.g. concerning which agents are allowed into

Content, or which messages to keep private and which to broadcast), as agents can easily have different tasks with different requirements.

Therefore, agents can *create external groups*, that is, an agent looking for others to help it achieve some goal can create a group, join it, and invite other agents that might be able to assist it to that group, rather its own content. That way, the inviting agent can have groups with different policies, a straightforward way to restrict messages to only reach relevant agents (namely agents within a certain group), yet it still is "in the loop", as it receives all messages that are sent between agents.

Last but not least, agents can *create internal groups*, that is, it can create a group agent within its *Content*, and invite agents to join that group[6]. The main difference between external and internal groups is that, in the latter case, the inviting agent effectively transfers control over the invited agents to the group — it does not receive communication occurring between group members, nor does it even know which agents joined the group. The inviting agent tells the group agent to solve a problem, and will be notified by the group once the problem is solved.

It should be noted here that the methods listed above present some issues that have to be solved. Firstly, whenever an agent invites others to join itself or a group, it does not know how many agents will actually react, or how long it takes the agents to join. Furthermore, agents might charge for their services, or provide solutions of different quality. While the above methods are still valid and have their respective advantages, agents will need some additional rules do decide when, and how, to choose between joining agents/services, be it through first-come-first-serve, auctions, negotiations, or other rules. Another matter that deserves close attention is group policies. In [8], several group policies specific to agents were introduced, ranging from groups where all agents can join and have no obligations to groups where only agents are allowed to join that agree to immediately honour requests from the group agent. Note that this, while similar to using different collection techniques, aims at a more fine grained level of control of the (group) agents.

4.6 Web Services

The idea behind web services is to have a collection of software services accessible via standardised protocols, whose functionality can be automatically discovered and collected together in order to provide an application working in a distributed way over the WWW. The difference between this approach and that of many (but not all) of the others above is that the services are meant to be found and grouped automatically. This search for appropriate services is supported by the idea of publishing a service's properties, for instance in XML, and then letting other agents/applets browse these published specifications in order to choose the most appropriate service [14]. Focusing on organisational matters, we can represent the initial user request as a transient group, that has to automatically find and integrate[7] a set of services to fulfil its initial goal. Once the goal is satisfied the group dies freeing all the *Content* services that it contains. Finally, in a

[6] Note that this is effectively implemented by interest groups (see Section 2.4).

[7] We disregard here all the challenges arising from a meaningful composition of services that are relevant to the Semantic Web and Ontology research communities.

number of cases, organisation of web services can be seen as corresponding to the organisation of agents, and so many of the suggestions provided for multi-agent systems above also apply to web services.

4.7 Exploiting the Group Structure

By now we have mentioned several distinct applications of the group approach, ranging from simulating object oriented behaviour over multi agent systems to feature interaction and web services. What makes our system so versatile is its ability to encode different types of meta information in the group structure. We can for example use the element structure to facilitate the communication between service robots and inhabitants of a building, by assigning a group element to each room, and grouping them in floors. Each robot joins the group of the room it is in at the moment. Furthermore, we assign agents to robots and to personal digital assistants (PDA) of the people working there. If some PDA requests coffee, the request will automatically be forwarded to the nearest robot. Also, if one robot needs help for opening the door, it will always address its nearest peer.

Other properties that can be represented in the structure are abilities of elements. The interest groups mentioned in Section 2.4, for example, create groups of elements that share the same interest / have the same abilities. We can again collect interest groups to allow agents to create hierarchical "yellow pages" that evolve within the system. Groups can also collect elements based on their ownership or affiliation.

Most obvious is the representation of tasks/subtasks through the structure. An often used example in agent technology is that of a travel agent that collects agents that deal with hotels, flights, and other aspects of a holiday. Those agents can, in turn, again make use of other, more specialised agents. Within a system where many of those travel agents are, the structure will represent the tasks that are required from them. Furthermore, defined hierarchical structures, such as in organisations and also in object oriented programming, can easily be represented.

One should note that above examples are by no means exclusive — as elements can be part of more groups, we can have several of above mentioned structures superimposed on one system. As the aims and goals of each structure may be quite disjoint from the others, it should be straightforward for elements to distinguish internally between the different types, and make full use of them.

5 Summary

In this chapter, we have provided a novel model for organising computation using group structures. These structures are both flexible and powerful. Further, since the basic framework is implemented in Java, then the computational elements can take a wide variety of forms. For example, if we consider basic Java, the elements are simply objects and so the grouping structures can provide object-oriented structures, component architectures, etc. If the elements concerned are *agents*, then grouping provides mechanisms for organising multi-agent systems, for example *teams*. And so on. By examining such instances of the model, we hope to convey the generality and flexibility of the approach. By identifying elements and groups as essentially the same entities, the model

becomes particularly simple (with correspondingly simple semantics, described elsewhere [5]). Consequently, this lightweight layer can be added to many types of system in order to provide powerful organisational capabilities.

References

1. Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, May 2001.
2. Kenneth P. Birman. The Process Group Approach to Reliable Distributed Computing. Techanical Report TR91-1216, Department of Computer Science, Cornell University, July 1991.
3. G. Booch. *Object-Oriented Analysis And Design With Applications*. Addison Wesley, 1994.
4. M. Calder, M. Kolberg, E.H. Magill, and S. Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Computer Networks*, 41(1):115–141, 2003.
5. M. Fisher and T. Kakoudakis. Flexible Agent Grouping in Executable Temporal Logic. In *Proceedings of Twelfth International Symposium on Languages for Intensional Programming (ISLIP)*. World Scientific Press, 1999.
6. S. Franklin and A. Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In J. P. Müller, M. J. Wooldridge, and N. R. Jennings, editors, *Intelligent Agents III — Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages (ATAL-96)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Heidelberg, 1996.
7. Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.
8. B. Hirsch, M. Fisher, and C. Ghidini. Organising logic-based agents. In M.G. Hinchey, J.L. Rash, W.F. Truszkowski, C. Rouff, and D. Gordon-Spears, editors, *Formal Approaches to Agent-Based Systems. Second International Workshop, FAABS 2002*, volume 2699 of *LNAI*, pages 15–27. Springer, 2003.
9. Kate Keahey. Common Component Architecture Terms and Definitions, Common Component Architecture Forum. `http://www.acl.lanl.gov/cca/terms.html`.
10. M. Plath and M. D. Ryan. Feature Integration using a Feature Construct. *Science of Computer Programming*, 41(1):53–84, 2001.
11. C. Szyperski. *Component Software - Beyond Object Oriented Programming*. Addison Wesley, 1998.
12. D. Ungar, C. Chambers, B-W. Chang, and U. Hölzle. Organizing Programs Without Classes. *Lisp and Symbolic Computation*, 4(3):37–56, 1991.
13. W3C consortium. W3C Web Services Glossary. `http://www.w3.org/TR/ws-gloss`.
14. W3C consortium. W3C Web Services Activity Statement, 2002. `http://www.w3.org/2002/ws/Activity`.
15. Peter Wegner. Classification in object oriented systems. *ACM SIGPLAN Notices*, 21(10):173–182, October 1986.
16. M. Wooldridge. *An Introduction to Multiagent Systems*. John Wiley & Sons, 2002.
17. M. Wooldridge and N. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.