Object Naming and Object Composition

D. Ramazani, G. v. Bochmann and P. Flocchini

Publication #1135

Département d'informatique et de recherche opérationnelle

Université de Montréal

Novembre 1998

Object Naming and Object Composition

D. Ramazani, Université de Montréal
P. Flocchini, Université du Québec à Hull
G. v. Bochmann, Université d'Ottawa

Abstract

In this paper, we show that a flat domain of object names is inappropriate in the context of object composition since new naming requirements must be fulfilled. These requirements can be summarized by: (1) visibility and hiding of components, (2) checking whether two objects belong to the same composition, (3) sharing of components between compositions, (4) reference from one composition to another composition. Object naming in the object paradigm is based on the use of the concept of object identity which can be considered as an intrinsic and universal name of the object. It does not record information about the structure of objects. This naming mechanism is inadequate for fulfilling the requirements stated above. For instance, once you have the identity of an object you can access that object. This means you can not achieve component hiding within the composition.

There are many alternatives for solving this problem including the design of complex naming mechanisms. Our solution consists of recognizing that referencing objects is contextual while the concept of object identity is context independent. Based on this observation, we view the execution of an application as an evolving structure of labelled graphs of objects. Each state of the application is represented by a graph of objects. This allows us to use the concept of sense of direction [Floc98a] for designing various kinds of naming mechanisms.

In this paper, we designed three naming mechanisms using sense of direction. The first is based on a chordal sense of direction. The second is hierarchical naming without component sharing. The last is a slight modification of the second to allow component sharing. This is achieved by introducing special names for shared components.

Keywords: Composition, Naming, Object identity, Sense of Direction.

1. Introduction

Designing object-oriented distributed applications consists of defining precisely collective and individual behaviors as well as the states of the objects forming the application. The way we describe interactions between these objects plays a significant role in the specification of the behavior of objects; ultimately, it influences how we implement applications. Interactions between objects require that at least some of these objects be referred to. This is achieved by means of an object naming mechanism. Such a mechanism affects three aspects of the application, namely :
1) the definition of object interfaces: it determines how we hide certain details of objects;
2) the algorithms that implement the object operations, as well as the data structures used for representing the object state;
3) object communication with one another, as well as the representation of associations between objects.

In many object-oriented languages, the concept of object identity is used as an object name. Since object identities are intrinsic properties of objects, this corresponds to the use of global names for objects. The obvious implementation for the concept of object identity is a pointer (or reference) which represents a memory address. In the distributed context, this can be complemented with network addresses and task identifiers. As mentioned above, an object naming mechanism plays a significant role in the description of the collective and individual behaviors as well as the states of objects. In this context, one may ask if global names are still appropriate for applications with composite objects. Intuitively, the linkage between the structure and the behavior of composite objects established in [Rama95a] suggests that this structure will influence the way objects communicate. This will in turn impact on how we designate and reference the communicating objects.

1.1. Summary of our experience

In this paper, we show that global names are inappropriate for applications having composite objects since new naming requirements must be fulfilled in these applications. These requirements include : (1) visibility and hiding of components, (2) checking whether two objects belong to the same composition, (3) sharing of components between compositions, (4) reference from one composition to another composition. These naming requirements are not totally fulfilled by the existing object naming mechanisms, especially those based on the concept of object identity as unique name provided by an object. For instance, if we consider the necessity to hide some components, object naming in the object paradigm is such that once you have the identity of an object you can access that object. This means that, if a component before its insertion into the composite object was known to an object outside the composition, the latter object can still access the component even after its insertion within the composition as a hidden component. There is no means for preventing such an access except by removing all the references from objects outside the composition to the component before its insertion within the composition. This operation virtually implies the access to all objects. On the other hand, if we want to check if two objects belong to the same composition, this can not be achieved by using the identity of these objects since the identity of an object has no record of the context in which the object is located, nor it records the structure of the objects.

There are many alternatives for solving this problem including the design of complex naming mechanisms. Our solution is simple and practical, it consists to recognize that referencing objects is contextual while the concept of object identity is context independent. Based on this observation, we view the execution of an application as an evolving structure of labelled graphs of objects. Each state of the application is represented by a graph of objects. Viewing the state of an application as a graph of objects allows us to use the concept of sense of direction [Floc98a] for designing various kinds of naming mechanisms. Sense of direction exhibits many properties that allow the design of naming mechanisms which can: distinguish between objects, permit the transfer of object names between objects, handle sharing of components, and preserve the locality of names. We later show how many properties of sense of direction relate to the properties of naming mechanisms. In this paper, we describe three naming mechanisms using sense of direction. The first is based on a chordal sense of direction. The second is hierarchical naming without component sharing. The last is a slight modification of the first to allow component sharing, this is achieved by introducing special names for shared components.

We have considered naming based on a chordal sense of direction. It is based on a chordal labeling of the graph of objects which is defined by fixing some arbitrary cyclic ordering of the nodes and labeling each incident edge by the distance in the above cycle. This means for a given object system, the number of objects forming this system is used for fixing the cyclic ordering of the nodes, and local names attributed by one object to the objects it knows are based on this number. In addition, for this chordal labeling to be a sense of direction, the local names attributed by one object to the other objects must be relative distances in the cyclic ordering [Floc98a]. The main difference with hierarchical naming is that the naming is not linked to the structure of the application and its objects.

Our naming scheme based on hierarchies of objects considers the application as being a composition of objects which can later be refined into other compositions. At the initial level, there are the application and its objects. The application has a naming context which attributes names to the objects, i.e. a naming context is a mapping between names and objects. In a naming context, names are unambiguous. In addition, objects may be refined into compositions. For each composition, there is an associated naming context. Components of a given composition have local names in that composition. They may refer to each other using these local names. An object of one composition may reference an object of another composition through the compositions, i.e. reference between compositions requires a composite name which includes the names of the compositions which are involved in this reference (naming action). This simple mechanism allows to take care of many of the naming requirements except component sharing.

In these experiences, we found that allowing component sharing complicates a lot the naming mechanisms. This raises the question of the necessity of allowing component sharing between compositions. From a modeling point of view, compositions which share components might be considered as a single composition. Therefore, component sharing is a modeling artifact which can be ignored. This is the approach taken by many authors [Rumb94].

1.2. Organization of this paper

It will be pretentious to provide in this paper an in depth analysis and a complete account of all the aspects of object naming in the context of composite objects. The reader is referred to [Wier94, Kent91] for other aspects of object naming. Therefore, we shall purposely focus on the naming mechanisms we propose with regard to the new requirements in the context of object composition.

The paper is organized as follows. Section 2 introduces the basic concepts of object naming which are relevant for understanding the rest of this paper. Next, Section 3 describes the new requirements of object naming imposed by object composition. Section 4 introduces the concept of sense of direction and its application to naming mechanisms. It describes how chordal naming and hierarchical naming can be used in the context of object composition. In addition, it describes related work. Section 5 concludes the paper and describes future research.

2. Object Naming

2.1. Characterizing object naming

Many aspects characterize object naming [Kent91, Wier94]. First are the syntax and the semantics of object names. Generally, name syntaxes are implementation dependent. For example, object names may be user defined, e.g. containment relationships may be taken into account when attributing names to objects. On the other hand, object names may be system generated. Furthermore, in the context of object databases and distributed systems, we may distinguish physical object names which are implementation dependent from logical object names which tend to be independent of any implementation. For example, the concept of object identity has been used as a special name provided by each object. In object-oriented notations and programming languages, the aim of object identity is to provide a means for distinguishing between objects independently of their states and/or behaviors.

Apart from its distinguishing capability, the semantics of an object name may embed other properties. For instance, in the context of object databases and distributed systems, the current location of an object (e.g. within the database, within the application cache, or within the application memory) may be encoded in its name. Access control and object locking information may also be encoded along or within the name of an object.

Both the syntax and the semantics of names are grouped into a single concept, namely the naming convention which stands for the syntactic representation of names as well as their semantic interpretation. A name space or domain is a set of names complying with a given naming convention. The operations allowed on object names are performed in the context of a naming mechanism. Object names are used to refer to objects; to provide information about those objects (e.g. type information); to locate objects given only their names; and to access those objects. All these functionalities of an object naming mechanism are based on the operations one may perform on object names. Among these operations, we find comparison, assignment of a name to an object (also known as binding), resolution (i.e. finding the object behind the name), etc.

In addition to these properties, Wieringa and de Jonge [Wier94] require that an object naming mechanism should be monotonic. An object naming mechanism is monotonic when from one state of the application to another :

1) the name refers to the same object;

2) the object remains named by the same name.

This precludes object renaming and reuse of object names.

2.2. Structure of the set of object names

The set of object names may be flat or structured. By structured, we mean it is partitioned according to some criteria. For example, some authors propose to structure object names based on the hierarchical relationships that exist between the named objects. Others propose to partition the set of object names into contexts or spaces defining the validity of the name. By a valid name, we mean a name which can be successfully resolved.

Generally, when the set of object names is structured, this is reflected on the syntax and the semantics of object names. Flat sets of object names go along with flat (or primitive) object names. Structured sets of object names may imply partitioned object names which in turn are commonly structured as a series of primitive object names. In addition, the names can be global or local. When the set of names is partitioned into sets, these sets define the validity of names. A local name is only valid in a given set of names, while a global name is valid in any set of names with respect to the naming convention considered.

2.3. Object identity and smart pointers

As explained before, in the object paradigm, the concept of object identity is used for naming objects. The object's identity can be considered as a global name provided by the object. However, in many ways, the concept of object identity does not fulfill the needs of application designers. As a matter of fact, the advent of smart pointers in C++ illustrates the need for other kinds of object names [Meye96]. In other words, object identity is inadequate as an object name since it conveys only the information that allows to distinguish between objects and no more. In C++, smart pointers are objects that are designed to look, act, and feel like built-in pointers (i.e. object names), but offer greater functionality. In particular, they allow to gain control on:

• construction and destruction of pointers to objects;
• copying and assignment.
• dereferencing (pointer resolution), for instance in database applications, you may decide to bring the object in memory. The smart pointer transparently handles the object location.

As a matter of fact, the object database standard relies on the use of smart pointers to transparently access transient and persistent objects in database applications. With smart pointers, software designers express the insufficiencies of the object naming mechanism provided by C++, i.e. the use of the concept of object identity as an object name. This naming mechanism conforms to what is expected with the object paradigm. Is this the plea that we need changes to the object paradigm with respect to object naming? We believe so, because with the use of smart pointers, we are altering the way we perceive and understand object names. Smart pointers may add access control and other information to the object names. In addition, smart pointers may reflect the organization of the set of objects.

3. Operations on names

In this Section, we describe the various operations which can be executed on object names. These operations are classified into general operations, and operations specific to object composition. Once the syntax and semantics of names are defined, it should be stated how the names are used in specification, design and programming of applications. The operations on names reported in this section cover these steps of application development.

3.1. General operations on names

Four generic operations can be defined on names, namely attribution, comparison, resolution, and communication. Attribution denotes the operation through which a name is associated to an object, i.e. the object is named. This operation is sometimes referred to as binding a name to an object. It occurs when an object is created or when another object wants to reference the latter object using a name it assigns to that object. When global naming is used, object names are usually system generated, and objects usually have unique names. When local naming is used, an object may have multiple names according to the objects which are referencing this latter object.

Once a name is attributed to an object, it can be used to define the semantics of object actions. For instance, one object may want to know if two objects are equal. This can be achieved by comparing the names of these two objects. Comparison of names is based on the equivalence relation defined between names. This relation assumes that two names belong to the same equivalence class when they denote the same object. This relation is used for defining a comparison relation between names. When global naming is used, such an equivalence relation between names is easy to implement since it consists of partitioning the set of names. However, when local naming is used, comparing names can be cumbersome. We will see later how it can be achieved in the context of local naming.

In addition to comparing names, an object may want to access to another object through its name. For instance, it may want to query the value of an attribute of the a given object, or it may want to call an operation of a given object. To achieve that, the object should be able to find an object by using its name. This is called name resolution, it means having access to an object through its name. In this case, when global naming is used, it is implemented by means of a mapping between names and objects. When local naming is used, since an object may have multiple names, resolving a name can be a non-trivial task.

Objects may also exchange information about other objects. This can be achieved by exchanging the names of objects. This means sending to a given object the name of another object, even its own name. To that extent, the receiving object must be able to understand the name which is communicated to it. This corresponds to communication of names. When global naming is used, every object is able to understand the names it receives, they are communicated as they are. In the context of local naming, names understood by one object may not be by another object. This requires translation of names so that the latter object may now understand the name which is received.

3.2. Issues related to object composition

Visibility and hiding of components

When considering a composition, the objects that belong to the composition are the component objects and the objects which do not belong to the composition are the objects outside the composition. Visible components are the components whose names can be exported to objects outside the composition. Hidden components are the components that must remain unknown to objects outside the composition. Component hiding implies the following restrictions:

R1: An object not belonging to a given composition may not interact with a hidden component .

R2: There is no restriction on a hidden component to interact with other objects outside the composition.

Visibility and hiding of components requires that we clarify how we reference a component from another component, from the composite object, and from an object outside the composition. Such a reference operation may use global names or local names. For instance, since in the object paradigm, the object identity is used as a name of the object, this corresponds to global naming. Another naming mechanism may use local names which in turn may be role-based or not. In a composition, the names of components are role-based when they are relative to the role played by the component in the composition. The names of components are not role-based when they are based on their identity. For instance, if we substitute a component in this context, we should use another name to reference the substituting component, while in the former case, the old name is still meaningful.

Once the mechanism used to reference the components is known, we must decide how we make components visible or hidden. In fact, there are many advantages to achieve visibility or hiding by including this capability into the naming mechanism. When the names are resolved, we can deny or grant the access to an object. This means each time an interaction has to take place between two objects, the object naming mechanism must make sure that such an interaction is desirable. This supposes that the object naming mechanism must be aware of the componenthood of the interacting objects. We require that the restrictions related to hiding of object must be enforced by the object naming mechanism since they can not be statically checked. In fact, visibility and hiding of objects, when the objects are manipulated through their names, can be viewed as an extension of the name resolution mechanism, i.e. when names are resolved, we can deny or grant the access to an object based on the object using the name and the target object.

Checking if two objects belong to the same composition

It may be useful to know if two objects belong to the same composition. This will ease the design of algorithms dealing with recursive data structures. With composition, we may take advantage of the preorder relation between objects based on the hierarchical relationships between the objects. For instance, when we are copying a hierarchy of objects, it is important to know if an object was already copied based on the composition to which it belongs. In addition, many distributed applications are based on hierarchies. For instance, network management assumes that the network is made of various components which in turn are made of other components. The failure of one component may cause the failure of the composition and also the malfunctioning of the other components. In such a context, we are interested to know whether a faulty network card is in the same equipment as a given communication port. Checking whether two objects belong to the same composition is an operation which can be seen as an extension of the operation of comparing object's names.

Checking if one object is a component of another object is a variant of checking whether two objects belong to the same composition. One of the best approaches to checking whether two objects belong to the same composition should be based on operating at the level of the object names. This suggests that the object name records the information about the object hierarchy. This can not be achieved when object identities are used as object names. In addition, global names are independent of the organization of objects in the application. This means, global names can not record information about the object hierarchy. This leaves as only alternative to use local names.

Sharing of components between compositions

From the modeling point of view, we may be interested to model compositions which share components. There is no particular problem with component sharing when we use global naming, especially object identities, to name objects. However, when local naming is used and objects are named on a per composition basis, component sharing implies that a component may have more than a name since it may belong to more than one composition. How can we check that two names resolve to the same object? In addition, how does the shared component resolve the names of other objects. This is a matter of concerns since two components of different compositions may have the same name.

Referencing objects between compositions

Interactions between objects are not limited to objects belonging to the same composition. Objects belonging to different compositions may also communicate. This means the object in one composition should be able to reference an object in another composition. When global naming is used, there is no particular problem. However, with local naming, and in particular when names are attributed on a per composition basis, it is not clear how an object in a given composition may reference an object in another composition. In this context, composition of names may allow us to cope with reference between compositions, i.e. the objects in other compositions are refered to using a combination of the name of the composition and the local name of the object in the composition.

3.3. Alternatives to the object naming in the object paradigm

In this section, we have exposed various issues which need to be addressed with respect to naming in the context of object composition. These issues can be considered as additional naming requirements which must be fulfilled by any naming mechanism which has to deal adequately with object composition. In order to fulfill these new naming requirements, we may change the way we interpret object naming. A shift in our thinking could consist of viewing an object name as a channel which allows to communicate with an object. Only communication paths between objects are named and not the objects themselves. Or, we could recognize that the new naming requirements can not be fulfilled by the available object naming mechanism, since they represent orthogonal requirements to that actually fulfilled by the concept of object identity. As a consequence, a different concept may be more adequate for tackling these new naming requirements. For instance, we may use different concepts for different purposes, i.e. the identity to distinguish intrinsically one object from another, and (contextual) names to reference the objects. In addition, object identity is not contextual because the concept of object identity is aimed at distinguishing the objects independently of their use. However, object reference is contextual. It implies a context, a lifetime, a relationship, and a meaning. As a consequence this context can be used to find out the object denoted by a name.

A variety of alternatives are at our disposal for fulfilling these new naming requirements imposed by composite objects. We may adopt the "do not care" approach, ignoring these requirements and leaving each software designer with the trouble of devising specific modeling and implementation artifacts for coping with these naming requirements. Instead, we prefer to address the problem by proposing another approach to designating objects, even at the expense of changing the way we view object naming in the object paradigm.

It is interesting to notice that based on the syntax, semantics and operations on names, a graph remains a good abstraction for discussing about these concepts. The name syntax, semantics and operations can be described by means of graphs. For instance, assume a graph G=(E,N). Its nodes N represent the objects. The edges E represent the references between the objects. Assume also that the edges are labelled using labels at the start and at the end of the edge. The labeling convention represents the naming convention in the sense that when global naming is used, all the edges ending at a given object have the same labels. When the edges ending at the same node may have different labels according to the starting nodes, this corresponds to local naming. The syntax of labels corresponds to the syntax of names. In addition, one object may reference an object which is not directly reachable from it based on the graph G. Such references may be handled by adding a new edge to the graph, or by considering the sequence of edges leading to the referenced object from the referencing object. Operations on names are interpreted as follows. Attributing a name to an object consists of drawing an edge to that object. Comparing the names becomes an operation on the labels of edges and the nodes of the graph. Resolving a name is traversing the graph to the required object. Communication of names is interpreted as drawing a new edge from the receiving object to the object denoted by the received name.

On the other hand, operations on names in the context of object composition are interpreted as follows. Notice that these operations apply only on the hierarchical relationships between objects. Therefore, we may limit the graph of objects to a tree representing the hierarchical relationships between the objects. This tree is derived from the graph based on the assumption that each composition is able to reference its components and each component has a reference to the composition to which it belongs. This assumption is used to prune the graph with non-hierarchical references between the objects reducing the graph to a tree. Now, armed with this tree, we interpret the operations on names in the context of object composition as follows: visibility and hiding of components is linked to the way we traverse the tree. For instance, hidden objects can be tagged so that when we resolve a name, we can check if the object is visible or not from the standpoint of the object using that name. Checking if two objects belong to the same composition is achieved by operating on the tree. We shall see later how this interpretation of name syntax, semantics, and operations in terms of graph and graph operations leads to a formalization of naming.

4. An Approach to naming

4.1. Basic idea

We would like to handle the new requirements explained in Section 3 using a formal framework. In the past, naming mechanisms have been formalized with mathematical relations and partial functions, see [Wier94, Kent91] for more detail. Such a framework adequately captures the relationship between names and objects, and the issues related to the resolution of names, but it tends to be oriented towards global naming.

We have stressed that the new requirements require contextual reasoning for manipulating and attributing names. This has led us to model the state of an application as far as naming is concerned, as a labeled graph of objects. Instead of devising a completely new formal framework, we advocate the use of existing ones. Sense of direction, described in [Floc98a], appears to be a formal framework in which deep semantic issues of naming mechanisms can be addressed. Hereafter, we show how the new requirements can be described and fulfilled using the concept of sense of direction. One of the advantages of sense of direction is that we can study both, global and local naming. In addition, many properties of naming mechanisms can be described in terms of sense of direction. It is less low-level than graph theory, or mathematical relations, and partial functions.

In this paper, we use two kinds of sense of direction, one is based on hierarchical naming, and the other on a chordal naming. The reasons for adopting a hierarchical design in the context of object composition is largely that hierarchies are easy to understand and to implement. Hierarchy helps to ensure the uniqueness of names, to reduce ambiguity, and to impose common names for all objects.

4.2. Sense of Direction

Sense of direction is a property of labelled graphs which has been extensively studied in the context of distributed computing (e.g., [FRS96, Floc95, Floc97]; for a survey see [FlSa98]). It provides a framework in which answers to deep semantic questions concerning naming can be addressed, and solutions can be proposed based on semantic coherence and theoretical feasibility. As an example, communication of names may involve the transfer of a name to an object which interprets names in a different context than the sender. How can we prove that the naming mechanism is consistent with respect to the communication of names? We shall see later in this paper how this can be achieved in the formal framework provided by sense of direction.

Definition of Sense of Direction

Let G=(V, E) be a graph where nodes correspond to objects and edges to references between the objects. Let E(x) denotes the set of edges incident to node x. Every node associates a label to each incident edge. Let $\lambda_x(x,z)$ denote the label associated by x to the edge (x,z) in E(x). λ_x is a local edge labeling function such that $\lambda_x : E(x) \rightarrow \mathfrak{L}$ where $\mathfrak{L}$ is a set of labels. Let λ denote the set of all λ_x. We say that λ is a local orientation if each node can distinguish among its incident edges, i.e. $\forall\, x \in V, \forall\, e_1, e_2 \in E(x)$: $\lambda_x(e_1) = \lambda_x(e_2)$ iff $e_1 = e_2$.

A path in G is a sequence of edges in which the end point of one edge is the starting point of the next edge. Let P[x] denote the set of all paths with x as a starting point, and let P[x,y] denote the set of paths starting from node x and ending in node y. Let Λ be the extension of the labeling function λ from edges to paths. $\Lambda_x : P[x] \rightarrow \mathfrak{L}^+$ be the path-labeling function defined as follows: for every path $\pi \in P[x]$, $\Lambda_x(\pi) = (\lambda_x(x,x_1), \lambda_{x_1}(x_1,x_2), .., \lambda_{x_{m-1}}(x_{m-1},x_m))$ where $\pi = [(x,x_1), (x_1,x_2), .., (x_{m-1},x_m)]$.

Coding function

We now introduce the notion of coding and decoding functions. We also show how to use them in order to construct local names. A coding function is a function that maps a sequence of labels corresponding to a path in G into a value, in such a way that two sequences corresponding to different paths starting from the same node are mapped in the same value iff the ending point of the paths are the same.

More formally, a coding function f of a system (G, λ) is a function such that :

$\forall\, x,y,z \in V, \pi_1 \in P[x,y], \pi_2 \in P[x,z], f(\Lambda_x(\pi_1)) = f(\Lambda_x(\pi_2))$ iff $y = z$.

In other words, the coding function allows the node to understand, from the labels associated to the edges whether different paths from any given node x end in the same node or in different nodes.

Decoding function

A decoding function h for f is a function that given a label and a coding of a string (sequence of labels), returns the coding of the concatenation of the label and the string. It allows nodes to translate the local names of their neighbors.

More precisely, given a coding function f, a decoding function h for f is such that for all x, y, z $\in$ V, such that $(x,y) \in E(x)$ and $\pi \in P[y,z]$, we have $h(\lambda_x(x,z), f(\Lambda_y(\pi))) = f(\lambda_x(x,z) \,.\, \Lambda_x(\pi))$, where "**.**" is the concatenation operator.

Sense of direction [Floc98a]

A system (G, λ), has a sense of direction (*SD*) iff the following conditions hold:

1) λ is a local orientation,

2) there exists a coding function f,

3) there exists a decoding function h for f.

We shall also say that (f,h) is a sense of direction in (G, λ). Several examples of sense of directions (e.g., cartographic, chordal, neighboring) have been described in [Floc98a].

Naming scheme

In a given labelled graph G, we can define a local naming scheme as follows. Each node associates a local name to the other nodes in the system; let us denote by $\beta_x(y)$ the name that node x locally associates to y and by β the set of all β_x. The set $\{ \beta_x(y): y \in E(x)\}$ is also called local view of x. The collection of all local object naming schemes $\beta = \{ \beta_x: x \in V\}$ constitutes the object naming scheme.

In the case of a graph with sense of direction, an object naming scheme can be constructed based on the existence of the coding function f; more precisely, the name that an object gives to another object is either the label of the link between them (if there is such a direct link), or the coding of a sequence of labels leading to it. In other words, given a coding function f, a local object naming scheme β_x for x is constructed as follows:

$$\forall x,y: \beta_x(y) = \begin{cases} \lambda_x(x,y) \text{ if } y \in E(x) \\ f(\alpha) \text{ otherwise} \end{cases}$$

where α is the sequence of labels corresponding to an arbitrary path between x and y. Given a graph representing an object system, many different labelings could be constructed to have different senses of direction, each one providing a different naming scheme.

Sense of direction allows to define names which can be used in contexts where they are not defined. When the labeling is a sense of direction, it is possible to understand from the labels associated to the edges, whether different paths from any given node x end in the same node or in different nodes. Using this characteristic, we may distinguish between objects. In a labeled graph with sense of direction, there is a function which maps the sequences of labels associated to the paths from x to y to the local name $\beta_x(y)$ used by x to refer to y. This latter characteristic allows an object to refer to objects which are not directly reachable from it. If there is sense of direction, node x, based on the label $\lambda_x(x,y)$ and on the name $\beta_y(z)$, can deduce that the received information is about the node locally called $\beta_x(z)$. This allows objects to exchange valid object names, i.e. names which can be understood.

4.3. Relation between naming mechanisms and sense of direction

As observed in Section 3, a labelled graph is a good abstraction of a naming mechanism. This has led us to examine how the concept of sense of direction which is a property of labelled graphs is linked to naming. The concepts underlying sense of direction can be related the properties which characterize object naming mechanisms.

An object naming mechanism has two aspects, the naming convention and the operations on names. In a framework with sense of direction, the name syntax is captured by the edge-labeling function λ and its extension to paths Λ. The semantics of names are captured by the node-labeling function β. In particular, the name attributed by the node x to the object y is $\beta_x(y)$. On the other hand, the operations on names such as attribution, comparison, and resolution of names are captured by the node-labeling function and its properties. Communication of names is handled by the existence of a coding and a decoding function. This latter function is used to translate the names so that an object can understand the names it receives from the other objects. In the following, we illustrate how a chordal naming with sense of direction can be used as a naming framework.

Sense of direction has many properties which help to handle some of the issues related to object composition. For instance, checking whether two objects belong to the same composition can be achieved by exploiting the fact that the naming convention is a sense of direction. An algorithm based on message broadcasting in a graph, as explained in [Floc98b] can be devised. Since the naming convention is a sense of direction, the algorithm is guaranteed to complete in linear time over the entire graph. This is possible because with sense of direction, we can distinguish between two nodes based on their names, and the names known by one node can be understood by another node. Checking whether an object is a component of another composed object is a special case of checking whether two objects belong to the same composition. Depending on the naming convention, specific algorithms can be devised for answering to these questions as we shall see later in this paper.

With sense of direction, we can also handle visibility and hiding of objects in a composition. Recall that we have introduced this property as an extension of the operation of resolving a name based on the object using that name and the target object. It suffices to extend the node-labeling function such that it can now query a map defining the visibility of objects in compositions. In addition, communication of names between compositions is handled by imposing the translation of names before they can be used by another object. This is achieved through the decoding function h of the sense of direction. In fact, $\forall$ x, y, z: $\pi \in P[y,z]$, we have $f(\Lambda_y(\pi)) = \beta_y(z)$, and $\beta_x(z) = h(\lambda_x(x,y), f(\Lambda_y(\pi)))$, i.e. from $f(\Lambda_y(\pi))$ and $\lambda_x(x,y)$ we can deduce $\beta_x(z)$.

4.4. Exemplifying the use of sense of direction as a naming framework

Assume that each object uses its own local names to reference other objects. We also assume that an object uses different local names to refer to different objects. A chordal labeling of a graph G=(V,E), with $|V| = n$, is defined by fixing a cyclic ordering of the nodes and labeling each incident link (x,y) of x by the distance (modulo n) of the two nodes in the above cycle. More precisely, assume the graph G=(V,E), with $|V| = n$. Let γ: V->V be a successor function defining a cyclic ordering of G and let $\gamma^k(x) = \gamma^{k-1}(\gamma(x))$ for k > 0. Let δ: V^2 -> {0, .., n-1} be the corresponding distance function, i.e. $\delta(x,y)$ is the smallest k such that $\gamma^k(x) = y$. The labeling λ in G is defined such that $\lambda_x(x,y) = \delta(x,y)$. It can be easily shown that a graph with a chordal labeling has a sense of direction (see [Floch98a] for more detail). The coding function f is defined as follows

$\forall\pi \in P[x_0]$, $\pi = [(x_0,x_1), (x_1,x_2), .., (x_{m-1},x_m)]$:

$$f(\Lambda_{x_0}(\pi)) \quad = \sum_{i=0}^{m-1} \lambda_{x_i}(x_i, x_{i+1}) \bmod n$$

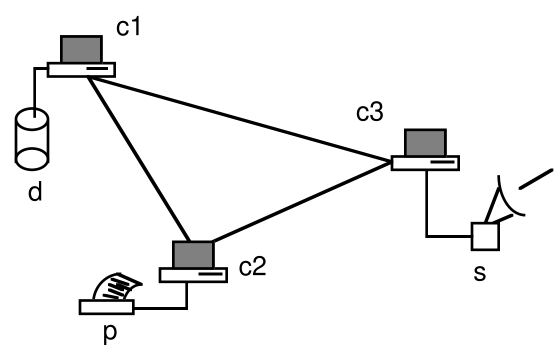
Its corresponding decoding function h is:

$\forall\ (x_0,y_0) \in E(x_0),\ \forall\ \pi \in P[y_0],\ \pi = [(y_0,y_1), (y_1,y_2), .., (y_{m-1},y_m)]$:

$h(\lambda_{x_0}(x_0,y), f(\Lambda_{y_0}(\pi))\ ) = \lambda_{x_0}(x_0,y_0) + f(\Lambda_{y_0}(\pi)) \bmod n$

To build an object naming scheme based on a chordal sense of direction we use the following. Given a coding function f, the local object naming scheme β_x for x is constructed as follows: $\forall$ x,y: $\beta_x(y) = f(\alpha)$ where α is the sequence of labels corresponding to a path between x and y. The collection of all local object naming schemes $\beta = \{\beta_x : x \in V\}$ constitutes the (global) object naming scheme. In this naming, we can not know if an object is a component of another object or if they belong to the same composition since chordal naming does not record the hierarchical relationships between objects. However, translation of names is fairly easy. For instance, when an object, say "x" wants to communicate to the object "y" the name of the object "z", this name can be translated. By definition of the decoding function h, $\forall\ (x,y) \in E(x)$ and $\pi \in P[y,z]$, $h(\lambda_x(x,z), f(\Lambda_y(\pi))) = \lambda_x(x,z) + f(\Lambda_y(\pi))$. By definition of the coding function f, $f(\Lambda_y(\pi))) = \beta_y(z)$, moreover $\lambda_x(x,z) + f(\Lambda_y(\pi)) = \beta_x(z)$. It follows that $\beta_x(z) = h(\lambda_x(x,z), \beta_y(z))$.

The figure below illustrates the application of this naming to a network consisting of three computers, namely c_1, c_2, and c_3. The computer c_1 is managing a large disk array "d"; c_2 is managing a laser printer "p"; and c_3 is managing a satellite dish "s".



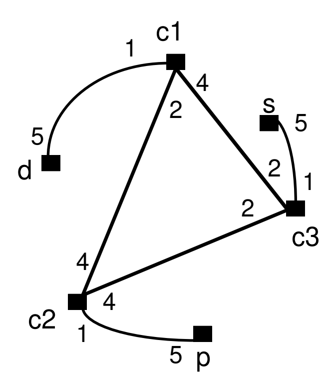


Figure 8.1: Chordal labeling of a computer network

In the graph with chordal labeling, the objects are represented by small squares. The local edge labels are used as follows. For instance, consider the object "c_1". When "c_1" references the object "d", it uses "1". When the object "d" references "c_1", it uses "5". When "c_1" references "c_2", it uses "2" and when "c_2" references "c_1", it uses "4", and so on. The translation of names occurs when the computer c_1 wants to send a document to the printer managed by the computer c_2. c_1 receives the name of the printer as "1" from the computer c_2. c_1 understands this name as $\beta_{c_1}(d)$ = "3".

4.5. Hierarchical Naming

In [Floc98b], it is shown that in a tree any labeling with local orientation is a sense of direction. Remember that a local orientation in a labelled graph is characterized by the property that a node uses different labels for its adjacent edges. This observation can be used for devising a hierarchical naming from the graph of objects. This is achieved by deriving a tree that reflects the hierarchical organization of the application. In particular, we introduce a root object which represents the application itself. In addition, we assume that there is no sharing of components between compositions. The object is created in the context of a given composition. The application itself is considered as a composition.

Nodes of the tree are compositions. Each composition has a local labeling function λ_x characterized by the labels in λ_x are from the set L where $L = L_0 \cup L_0^- \cup \{self\}$. The special label *self* denotes the composition. This means a node can reference itself using the label *self*. Elements of L_0 are written n, and $L_0^- = \{ n^- \mid n \in L_0\}$. n^- is the symmetric form of n in L. n and n^- are labels of edges linking two nodes x and y such that if x is a composition and y is a component of x, the edge (x,y) is labeled n and the edge (y,x) is labeled n^-.

As shown in [Floc98a], this labeling is a variant of a contracted sense of direction. The coding function f maps a sequence of labels corresponding to any path between two objects into the shoertest path between them. Here also, the name $\beta_x(y) = f(\alpha)$ where α is the sequence of labels representing a path between x and y. The figure below illustrates the computer network with this hierarchical labeling.

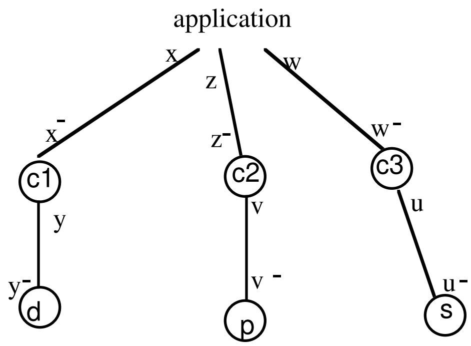


Figure 8.2: Example handled with hierarchical naming

As explained, the labeling of the graph may allow us to use simpler algorithms than one based on message broadcasting in a graph with sense of direction in order to check whether two objects belong to the same composition. Let $\beta_X(y)$, $\beta_X(z)$ be the names the object x attributes to the objects y and z with $x \neq y$, $y \neq z$, and $x \neq z$. The algorithm used by the object x to tell whether y and z belong to the same composition is based on the following properties. Keep in mind that we always assume that the application itself is a composition and all the objects are its components. This algorithm is applied at a given object x using the information provided by sense of direction. We assume tha each node is able to name the root of the hierarchy, i.e. $\beta_X(root)$ is known. From the object x, the objects y and z belong to the same composition iff:

• $\beta_X(root)$ is not a prefix of neither $\beta_X(y)$ nor $\beta_X(z)$;

or

• $\beta_X(root)$ is a prefix of both $\beta_X(y)$ and $\beta_X(z)$.

On the other hand, for the object x to tell whether y is a component of z, x can take advantage of the algorithm presented above because y and z have to belong to the same composition. Here, we have two cases:

(1) $\beta_X(y)$ has only labels in L_0^- and $\beta_X(z)$ has only labels in L_0. Here z is a component of y.

(2) $\beta_X(y)$ is a prefix of $\beta_X(z)$ and after cutting the prefix, the resulting $\beta_X(z)$ has labels only in L_0 or only in L_0^-. If $\beta_X(z)$ has labels only in L_0 then z is a component of y, otherwise y is a component of z.

4.6. Hierarchical naming with component sharing

To achieve sharing of components in the hierarchical naming proposed in this paper, we use the following artifact. We still consider the tree without component sharing. It is obtained by considering that objects have weight. Objects which are close to the root of the hierarchy are more weighted than those far from the root of the hierarchy. Based on that assumption, when a component is shared, it first belongs to the composition closest to the root of the hierarchy. This artifact allows us to derive a tree according to the previous hierarchical naming. Now, we complement this tree with chords (i.e. direct edges) linking a given composition to its components when such composition relationships are not portrayed by the previous tree. The label assigned to such a chord to the composition equals to the sequence of labels of a path between the two nodes linking the compositon to its component in the tree. Its symmetric is the inverse path.

This labeling of the hierarchy is also a sense of direction. Its coding function f maps a sequence of labels corresponding to any path between two objects into the shortest path between them. The naming function β is also defined to be equivalent to the coding function f. However, with such a naming, we are not able to answer to the question whether two objects belong to the same composition using the algorithm proposed for hierarchical naming. To answer this question, we use the algorithm which is based on depth-first search and message broadcasting in a graph with sense of direction as explained in subsection 4.3.

Finally, we have also explored the combination the hierarchical and chordal namings, into a single naming. The approach to their combination consists of structuring the application in terms of its compositions. This structuring is captured in a hierarchy of compositions. We use hierarchical naming for referencing one composition from another. Within a composition, we use a chordal naming. This is made possible by extending the node labeling function β to pairs of names, e.g. $\beta_x(y)=(a,b)$ where a is the name of the composition where y is located and b is the cyclic ordering of y within the composition. This results in a naming convention which has the advantages of both hierarchical and chordal namings. However, this naming is not intuitive.

4.6. Related work

A problematic similar to visibility and hiding of components have been studied by Hogg in the context of removing aliasing in object-oriented languages [Hogg91]. In a programming language with destructive read[1], Hogg introduces the idea of islands. An island consists of a container, i.e. set of objects, and a manager, which represents a the bridge to the rest of the world. An island is the transitive closure of a set of objects accessible from a bridge. A bridge is an object which can be considered as the container object for a set of objects. As such it is the unique access point to the objects in the container that make up the island; no object that is not a member of the island holds any reference to an object within the island except the bridge. An object outside the island can have a reference, by means of a variable with destructive read, to an object inside the island only through the bridge which manages the island. It can use the reference only once because of the mechanism of destructive read. This means if we associate a bridge to the composition, it can decides when to make available the references to its components, therefore enforcing visibility and hiding of components. Bridges and islands were originally devised as a technique for making proof systems for object-oriented languages practical. The solutions proposed are implementation oriented, i.e.

[1]Variables with destructive read are such that once they are read their values are gone.

they are described in the context of imperative object-oriented programming languages. They are not abstract enough to be useful in analysis and design of applications.

Chien and Dally have also studied the naming in the context of composite objects [Chien90]. They propose the concept of a concurrent aggregate to model a homogeneous collection of objects. A concurrent aggregate represents a group of objects (called representatives). Messages sent to the aggregate are directed to arbitrary representatives. The representatives of an aggregate can communicate by sending messages to one another. This is achieved through local naming within an aggregate. The representatives in an aggregate have indices from 0 to groupsize-1. groupsize is the number of representatives in an aggregate. The expression (sibling group 0) would return the name of the 0th representative in an aggregate. Representative names are ordinary object names or identities. Chien and Dally concentrate on intra-aggregate naming while we are concerned with more broader issues.

Local referential integrity introduced by Kappel and Schrefl is a concept which is in relation with composition of objects [Kapp92]. It attempts to apply the idea of referential integrity within the scope of a composite object. Referential integrity is satisfied if every object referenced exists. Accordingly, the concept of local referential integrity means that in any object reference, the referenced object and the referencing object belong to the same composite object. In addition, the authors mention that within their framework, objects are named using global and unique names which is proven in this paper to be inadequate for handling the new requirements in the context of object composition.

RM-ODP Naming [ODP97] is close to hierarchical naming as proposed in this paper. In addition, RM-ODP Naming introduces the concept of overlapping of naming contexts. When interpreted in the context of object composition it means having components which have the same names from one composition to another. However, RM-ODP Naming does not address the problem of checking if two objects belong to the same composition. In addition, the treatment of name communication is rather informal.

5. Conclusions

In this paper, we gave a new formulation of naming requirements in the context of object composition. These requirements can be summarized by: (1) visibility and hiding of components, (2) checking whether two objects belong to the same composition, (3) sharing of components between compositions, (4) from one composition referencing an object in another composition. We show that global names are inappropriate for fulfilling these requirements.

In an attempt to solve the problems related to these naming requirements, we designed naming mechanisms which are based on viewing the execution of an application as an evolving structure of a labeled graph of objects. Each state of the application is represented by such a graph. This allowed us to take advantage of the concept of sense of direction in the design of the different naming schemes. Sense of direction exhibits many properties that allow the design of naming mechanisms which can: distinguish between objects, permit the transfer of object names between objects, handle sharing of components, and preserve the locality of names. In this paper, we show how many properties of sense of direction relate to the properties of naming mechanisms. We describe three naming mechanisms using sense of direction. The first is based on a chordal sense of direction. The second is hierarchical naming without component sharing. The last is a slight modification of the first to allow component sharing, this is achieved by introducing special names for shared components. The main difference between chordal and hierarchical namings is that the hierarchical naming is not linked to the structure of the application and its objects. In these experiences, we found that allowing component sharing complicates a lot the hierarchical naming mechanisms. This raises the question of the necessity of allowing component sharing between compositions. From a modeling point of view, compositions which share components might be considered as a single composition.

Future developments of this work are concentrated in the area of formalization, especially the integration of the naming mechanisms which are proposed into specification languages. In this context, π-Calculus [Miln93] may play an important role. Since π-Calculus is built upon the notion of naming and because it provides a convient semantic substrate for object-oriented programming (see for [Miln93] more detail), the formal framework provided by sense of direction can be specified using π-Calculus, and later integrated in the formal semantics of an object-oriented programming language. Walker explored first the use of the π-Calculus to give semantics to concurrent object-oriented programming [Walk90]. In his formalization, he deals only with global names. We believe that sense of direction can be implemented as a naming discipline above the computational model provided by π-Calculus. Such a formalization will be more general than the one using sense of direction solely, because it will include the mechanisms of object interactions and the handling of multiple interfaces for objects. This formalization can later be used to describe the RM-ODP computational model. Future work of this research will explore the ideas sketched above.

References

[Chien90] Chien, A., Dally, W., Concurrent aggregate (ca), in Proceedings of Second Symposium on Principles and Practice of Parallel Programming, ACM, March 1990.

[Floc97] Flocchini, P., Mans, B., Santoro, N., On the impact of Sense of Direction on Message Complexity, Information Processing Letters, vol. 63, no 1, pp. 23-31 1997.

[Floc98a] Flocchini, P., Mans, B., Santoro, N., Sense of direction: formal definition and properties. Networks, Volume 32, No 3, pp.165-180, 1998.

[Floc98b] Flocchini, P., Santoro, N., Topological constraints for sense of direction, International Journal on Foundations of Computer Science, Vol. 9, No 2, pp.179-198, 1998.

[FlSa98] Flocchini, P., Santoro, N., Sense of direction in Distributed Computing, Proceedings of 12th international Symposium DISC '98, Andros, Springer-Verlag, LNCS, vol. 1499, pag. 1-15, 1998.

[Gosc91] Goscinski, A., Distributed Operating Systems: The Logical Design, Addison-Wesley Publishing Company Inc. 1991.

[Hogg91] Hogg, J., Islands: Aliasing Protection in Object-Oriented Languages, in Proceedings of OOPSLA'91, pp.271-285.

[Kapp92] Kappel, G., Schrefl, M., Local referential integrity, in Proceedings of 11th International Conference On the Entity-Relationship Approach, Karlsruhe, germany, October 1993, pp78-89.

[Kent91] Kent, W., A rigorous model of object reference, identity, and existence. Journal of Object-Oriented Programming, 4(3):28-36, June 1991.

[Lano93] Lano, K, Haughton, H., Approaches to object identity. In EROS II Workshop. LBMS London, 1993.

[Meye96] Meyers, S., More Effective C++: 32 New ways to Improve Your Programs and Designs, Addison-Wesley, 1996.

[Miln93] Milner, R., Elements of Interaction, Turing Award lecture, CACM Vol. 36, No. 1, January 1993, pp. 78-89.

[ODP97] ISO/IEC JTC 1, Information technology -- Open Distributed Processing -- Naming Framework, ISO/IEC DIS147771, July 1997.

[OLE93] Object Linking Embedding, Version 2.0, MicroSoft Release Notes and Technical Specs, 1993.

[OMG95] OMG, Naming Service Specification Clause 3, CORBA Services, OMG, March 1995.

[Rama95] Ramazani, D., Bochmann, G.v., A Conceptual Framework For Object Composition and Dynamic Behavior Description, Publication départementale #949, DIRO, Université de Montréal, Montréal, Canada, 1995.

[Rumb94] Rumbaugh, J., Building boxes: Composite objects, JOOP November/December 1994, Vol. 7, No. 7, pp. 12-22.

[Wier94] Wiering, R., de Jonge, W., Object identifiers, keys, and surrogates -- object identifiers revisited, Theory and Practice of Object Systems (October 94).

[X.500] Information Processing - Open Systems Internconnections - The Directory, ITU Recommendation X.500/ISO/IEC/IS 9594, Geneva, Switzerland, 1993.