7137

Using Objective-C

(for DFG manipulation)

Pieter J. Schoenmakers
Report

Using Objective-C (for DFG manipulation)

Report

by Pieter J. Schoenmakers

describing work performed
from Oct 1993 until Dec 1993
coached by dr. ir. Jos T.J. van Eijndhoven

Eindhoven University of Technology
Department of Electrical Engineering
Design Automation Section

This report was typeset by LaTeX in Palatino, Courier and Computer Modern.

This thesis describes DFGKit, a DFG Manipulation Library. DFGKit is available by anonymous FTP from `ftp.es.ele.tue.nl:/pub/tiggr/kit-`*xxx*`.tar.gz`, in which *xxx* denotes some version.

Abstract

This report describes a Data Flow Graph (DFG) manipulation library written in Objective-C. Since Objective-C is a rather unknown language to most people, a large part of the report is devoted to explaining how object oriented concepts apply to Objective-C and how it differs from C++.

C++ is used as the comparison language since it is a widely known language, which, like Objective-C is object oriented by definition. However, C++ differs radically from Objective-C, making for an excellent comparison: C++ follows the Simula 67 school of object oriented programming, where Objective-C follows the Smalltalk school. In C++ the static type of an object determines whether you can send it a message, in Objective-C the dynamic type determines it.

Throughout the whole report, C++ is observed as the language as it is used most often, i.e. using static typing. Furthermore, Objective-C is the language as implemented by NeXT and GNU, i.e. dynamically typed and missing static allocation.

Preface

This report describes a side effect of my graduation project, i.e. a DFG manipulation library that I wrote while writing a compiler to output DFGs. The report itself was a necessity since I had to give a presentation which, under the circumstances, could not do without a report accompanying it.

Because of severe time pressure this report is not what it could have been if I had had time to spend on it. However, after all, chapter 1 still gives a nice intro into Object Oriented concepts and chapter 2 provides a short but nice overview of Objective-C.

Acknowledgments

Thanks to my coach, Jos T.J. van Eijndhoven for proving to be flexible.

Eindhoven, December 15, 1993.

Tiggr

(tiggr@es.ele.tue.nl)

Contents

1

Introduction

This chapter is an introduction into the terminology concerning object oriented programming languages. The abstract definitions are completed by examples from C++ and Objective-C. Most of the information presented in this chapter is an excerpt from the Frequently Asked Question (FAQ) lists of the Usenet newsgroups comp.object [Hat93], comp.lang.c++ [Cli93] and comp.lang.objective-c [Sch93a], with additional details from [Woo93] and [NeX93].

The information presented in this chapter is meant to be neither exhaustive nor persuasive, though aspects to that effect will be implicitly present. This chapter merely contains an overview of concepts and terminology and of how it applies to both C++ and Objective-C

1.1 From non-OO to OO

Before object oriented programming was devised, two programming styles existed (which are of course still in use):

procedural
In a procedural programming language a global state is maintained through variables. Procedures and functions define the thread of execution, and each may modify any part of the global state.

functional
In a functional programming language, no global variables exist. The global state is implicitly present in the invocation path of functions.

Object oriented programming basically is an attempt to take abstract data types to a higher level of abstraction. Applied to the procedural programming paradigm, this implies that the global state is no longer maintained by publicly manipulatable variables. Instead, this state is a graph in which each node is an object which exhibits a certain behaviour. Messages can be sent to an object to evoke or manipulate its behaviour. Each message causes the invocation of a method (see figure 1.1). In the pure concept, the internal state of the object can not be modified in any other way.

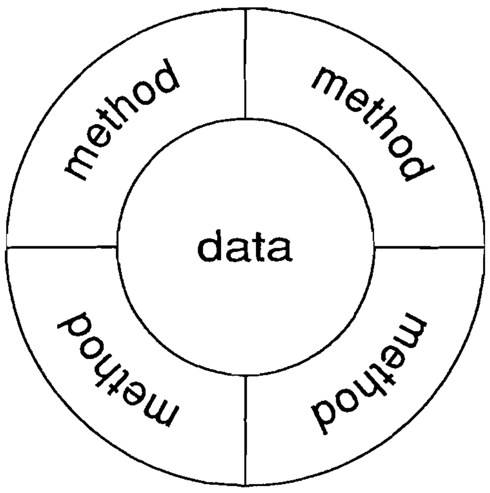

Figure 1.1: An object.

1.2 Classes

When modeling a system as a graph of interconnected objects, classification of those objects comes naturally. A *class* defines the behaviour and structure of its instances, which are objects of that class. Thus, an object defines state and is associated with a class which defines the structure of that state and the object's behaviour.

In object oriented programming languages, the following levels of classification can be discerned.

1 Classes do not exist. Only objects exist. An object defines state, structure and behaviour.

An example of a level 1 system is Self [US91]. In Self, sharing and inheriting behaviour (and state) is implemented through the notion of parent objects.

2 Each object is an instance of a class, but the classes are not available at run time. Each class merely exists as a concept at compile time to define the structure and behaviour of its instances at run time. This is applicable to C++.

3 From [Hat93]: Each object is an instance of a class and each class is an instance of the meta class.

Classes are available at run time and messages can be sent to them; the behaviour of a class is defined by the meta class. In this respect, a class is like an object.

This is almost applicable to Objective-C, with one notable difference: Each class is the sole instance of its own meta class. Furthermore, each meta class is an instance of the meta class of the 'Object' root class, thus avoiding the Infinite Meta-regress.

5 In a level 5 language, like Smalltalk, two extra levels are introduced to be able to make classes behave exactly like objects, i.e. a state whose structure and behaviour is defined by a meta class.

1.3 Inheritance

A class can be the subclass of another class. The latter is said to be the superclass of its subclasses. Subclassing is an important concept: A subclass inherits behaviour of a superclass. It may change this behaviour, or simply refine it. As part of this change, the subclass can extend the state to be held by its instances.

Subclassing encourages code re-use. Furthermore, it allows for higher levels of abstraction and introduces the notion of 'is-kind-of' predicates.

Using the hierarchy from figure 1.3.1, these concepts are all applicable:

abstraction
One can argue about mammals while dealing with cats and dogs.

is-kind-of
One can assert that a cat is a kind of mammal and that each mammal, including the cat, is a kind of organism.

code re-use
When talking about mammals, one can for instance be interested in the number of legs it has. This can be part of the state of a mammal; it need not be separately defined for cats and dogs.

refinement
When a cat encounters another cat, part of its behaviour will follow from the fact that the cat it is a mammal, though its behaviour towards the other cat will be different from the behaviour of a dog encountering the same cat.

In a level 1 language like Self, where classes per-se do not exist and objects define their own behaviour, inheritance is present at the level of objects instead of classes, as will be explained in section 1.4.

1.3.1 Single

Figure 1.3.1 shows a simple tree of classes. Since each class only has one parent class, this is called single inheritance. All level n ($n > 1$) languages provide at least single inheritance.

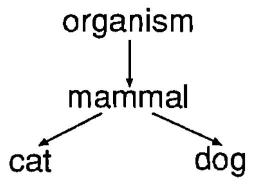

Figure 1.2: Single inheritance.

1.3.2 Multiple

When a class has more than one superclass, this is called multiple inheritance. The state held by an instance of the subclass is a union of the state of the superclasses (plus possible additions), and its behaviour is based on the union of the behaviour of all superclasses. The thief in figure 1.3.2 can exhibit criminal behaviour but can also be behave like a man.

Multiple inheritance implies more extensive re-use of code. It also implies polymorphism (see section 1.5).

C++ allows explicit multiple inheritance. Objective-C does not. According to some, this is a problem. According to others, different ways of obtaining the same result exist.

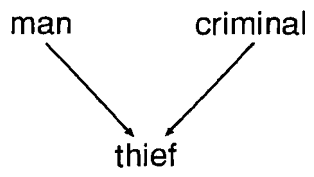

Figure 1.3: Multiple inheritance.

1.3.3 Shared

Shared inheritance solves a problem inherent to multiple inheritance. Consider the inheritance graph in figure 1.3.3. In this particular hierarchy, criminal is not just some behaviour, but a human exhibiting criminal behaviour. When making a class of male thieves, it is best described by the fact that each instance is not only a human criminal, but also a human male. With plain multiple inheritance, this would imply that a thief would incorporate the state of two humans, something which is not always desired.

A solution to this problem is shared inheritance: The state of a thief includes only one human state.

Shared inheritance is possible in C++, i.e., applied to the example, you have the choice to make every thief be one or two humans. One problem is that the sharedness of a superclass must be indicated in that class itself, instead of in the class desiring the shared behaviour. As with a lot of C++ constructs this requires clear-sightedness of the library designer.

Shared inheritance does not apply to Objective-C, since it does not provide multiple inheritance.

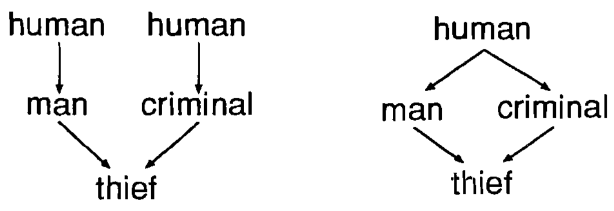

Figure 1.4: Multiple (left) v. shared inheritance (right).

1.3.4 Dynamic

Figure 1.3.4 depicts the concept of dynamic inheritance: An object can dynamically change class, and thus modify its behaviour. In the example shown, the adolescent suddenly changes into an adult, indeed a rare but possible event. Dynamic inheritance may also be used to change the parent classes of a class, affecting all future objects.

Both C++ and Objective-C lack explicit dynamic inheritance. However, in the following section, it will be shown that it is possible for an object to exhibit behaviour as if dynamic inheritance is applied to it.

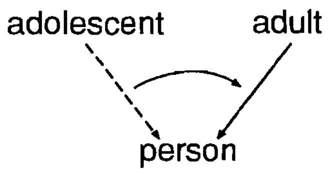

Figure 1.5: Dynamic inheritance.

1.4 Delegation

Figure 1.4 shows an object which is an instance of the class *Fighter*. Probably most of the time, this fighter will expose a fighter's behaviour. However, this fighter secretly knows a diplomat, i.e. an instance of the class *Diplomat*. The fighter can, if it decides to do so, invoke the services of the diplomat, in order to exhibit, to the outside world, diplomatic behaviour. Thus, other objects will observe that a fighter (sometimes) behaves like a diplomat. This concept of inheritance at the level of objects as opposed to the level of classes is called *delegation.*

Delegation can be used to make up for the lack of multiple inheritance in a class hierarchy, and is in fact often used in Objective-C. Since C++ does have multiple inheritance at the level of the class hierarchy, delegation is used much less in C++.

Since delegation delegates the implementation of a part of the behaviour of an object to an instance of a different class, it can also be used to mimic dynamic inheritance. Note however, that this possibility is rather restricted in statically typed languages like C++.

In Self, a level 1 language, an object is self describing. However, it can explicitly set a *parent* object which is to deliver behaviour that is shared among several objects. This is, of course, a special form of delegation.

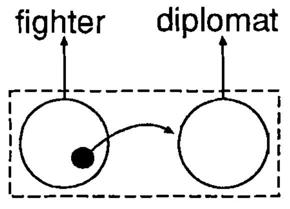

Figure 1.6: Delegation.

Delegation has some advantages over multiple inheritance (as implemented by C++): When recompilation of a class *super* causes the layout of its instances to change, any class inheriting from *super* also needs recompilation. This is especially nasty if *super* is part of a library. When using delegation, any class using instances of *super* as a delegate do not need recompilation. Any subclass of *super* still does, of course.

1.5 Polymorphism

Polymorphism refers to the ability to appear in different forms. Using the single inheritance example from figure 1.3.1, a cat, when treated like a mammal, will actually behave like a mammal. However, when taking a closer look, it becomes clear that it is not just another mammal, but actually a cat.

Looking at this example of polymorphism, it is clear that the concept of polymorphism is present in both C++ and Objective-C. However, C++ imposes a severe limitation on polymorphism, which is inherent to its typing scheme, as explained in the following subsection.

1.5.1 Static typing

C++ uses *static typing*. Static typing means that the type of an object must be known at compile time. Since it is the type of an object that defines what messages can be sent to it, only messages which are known to be acceptable at compile time can be sent. Unless the programmer would revert to inclusion polymorphism [Hat93], i.e. automated re-use of *source* code, this restriction implies that only instances of subclasses of a certain base class can act like instances of that base class.

1.5.2 Binding

Static typing poses a problem with the level of refinement that a subclass can add to its base class. If a method of an instance of a subclass is invoked, which is implemented by the super class, the code in the method will think about the object as being an instance of the superclass. However, when invoking another method of itself, this will cause the method of the superclass to be invoked, even though the subclass actually overrides this method (figure 1.7).

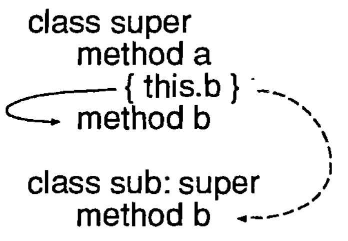

Figure 1.7: Static binding.

A solution to this problem is so-called statically typed dynamic binding. In the previous paragraphs, only static binding has been applied. In C++, it is possible to define certain methods, or member functions in C++ speak, to be virtual. The net effect of this is that the actual method to be invoked will be determined at run time, instead of being derived from the type of the object at compile time.

Statically typed dynamic binding is an attempt to solve the refinement problem, but it is not an actual solution. The (unsolvable) problem is that it is up to the designer of the superclass which methods are to be dynamically bound. This clearly is a problem for somebody who wants to refine the behaviour of a class for which the source code is unavailable.

1.5.3 Dynamic typing

Dynamic typing is the ability to, given an object, determine its type, *at run time*. In a dynamically typed language, dynamic binding comes naturally. Objective-C is an example of a dynamically typed language. Some dynamically typed environments do allow static binding, but this is of course a useless restriction in flexibility.

One implication of dynamic typing is that polymorphism is not restricted to subclasses, as in C++. Any object, of any class, which decides to behave like another object can actually do so.

2

Objective-C

This chapter will discuss some of the features of Objective-C and discuss common problems, or issues often perceived as problems.

2.1 Feature summary

Objective-C is a superset of ANSI C, adding objects and classes in a Smalltalk fashion. The following object orientation paradigms apply to Objective-C:

level 3 language

Objective-C is a level 3 language: Each object is an instance of a certain class and each class is the sole instance of its meta class. To avoid the infinite meta-regress, each meta-class is an instance of the meta class of the Object class.

This setup implies that classes can be handled just like normal objects: one can send messages to them. On the other hand, each class holds exactly the same state as each other class, implying that so-called 'class variables' are not available.

single inheritance

The Objective-C class hierarchy is a tree. At the root of the tree resides the Object class. This class is responsible for the creation and destruction of objects and is needed for the runtime to be able to do its work.

dynamic typing

Objective-C is a dynamically typed language. This means that full typing information on each object is present at run time. It also implies that method binding is always dynamic.

Dynamic typing causes all objects to be equal at compile time. Therefore, a pointer to an object, irrespective of its class, is almost always an 'id'.

dynamic binding

All method binding in Objective-C is dynamic. Matching is based on the textual name of a method. This name is called the selector. A selector does not contain any typing information, making the selector for the following two methods equal: '-(int) value' and '-(float) value'. In both cases, the selector is 'value'.

In case the compiler has seen both method declarations, sending the message 'value' to an object whose declared type is 'id' causes a problem in that the compiler can not know what return value to expect from the method invocation. In this case, the static type of the object can be indicated to help the compiler resolve the method to invoke. However, the actual binding will still be performed at run time.

delegation
Delegation is an often-used concept in circumstances where some might prefer multiple inheritance. Delegation and other concepts which make up for the lack of multiple inheritance are discussed in section 2.2.

encapsulation
All methods in Objective-C are public. Instance variables can be hidden from prying eyes by declaring them to be public, private or protected, much like in C++.

2.2 Multiple inheritance issues

Objective-C does not provide multiple inheritance. According to some, this is a problem. However, there are several rather elegant solutions to this problem.

2.2.1 Delegation

Delegation is a simple technique where an object has a pointer to another object and sometimes has that object operate on its behalf by forwarding messages to it. After all, object orientation is about objects exhibiting a certain behaviour by responding by method invocations.

One way method forwarding can be implemented is on a per-method basis, by implementing each method to be forwarded to the delegate:

```
-method: argument
{
  return ([delegate method: argument]);
}
```

Another solution is to use 'forward::' and 'performv::' to forward anything or just a group of methods to a delegate. Below is an example where only specific methods are send to delegate1, whereas anything else is forwarded to delegate2.

```
-forward: (SEL) aSelector : (arglist_t) arguments
{
  if (aSelector == @selector (method1)
      || aSelector == @selector (method2))
    [delegate1 performv: aSelector : arguments];
  else
    [delegate2 performv: aSelector : arguments];
}
```

2.2.2 Protocols

Protocols provide an elegant solution to polymorphism problem which multiple inheritance tries to solve. Multiple inheritance is mostly used to ensure that a certain class responds to some set of methods. This is exactly what protocols also do: A protocol is a definition of a set of methods. A class can be said to adhere to a set of protocols. If, at compile time, the compiler discovers that a class does not implement all methods of a certain protocol, it will issue a warning, giving control over the implementation of a protocol by a class in case a protocol changes.

Another application of protocols is to help the compiler resolve selector names by declaring that an object pointed to adheres to a protocol:

```
/* A pointer to any object.  */
id o1;
/* A pointer to an object that adheres
   to the TableElement protocol.  */
id <TableElement> o2;
```

Of course, protocols do not in any way ease code re-use, a concept that can be realised when using multiple inheritance.

In short, the application of protocols equals multiple inheritance in case all but one of the superclasses are an abstract class, i.e. a class which does not define instance variables, which does declare methods but which does not implement them. Put differently, adherence to a protocol is a promise to exhibit some specific behaviour in that instances of the class respond to a specific set of messages.

2.3 Garbage collection

A often-heard complaint from programmers with a Lisp, CLOS or Smalltalk background, which are new to Objective-C is that it does not provide garbage collection (GC). This is, of course, a mere display of ignorance, since it can be easily applied to Objective-C. Currently there are two ways of gaving garbage collection:

1. Using the possibilities offered by the software described in [BW88]. This software implements garbage collection on memory allocated through its replacement of malloc. Since it normally is malloc that is used to allocate objects, both in C++ and Objective-C, objects that are no longer referenced will be forgotten. Note that modification of the library is required if the garbaged objects need to be sent some kind of notification of their state.
2. Using the possibilities offered by the LispObject class provided by [Sch93b]. This class override's '+alloc' and performs garbage collection on objects created this way. Objects take an active part in garbage collection by referencing objects they know during the first phase of the mark-and-sweep algorithm used by the garbage collector. After this phase, each object that has turned into garbage is sent a message to tell it that it is to be deleted.

The DFG Manipulation Library uses the second GC implementation.

2.4 Speed comparison

An often heard complaint about dynamically typed languages is that the method binding is too slow. At least for Objective-C with the GNU Objective-C runtime introduced in GCC 2.4, this statement simply isn't true.

Table 2.1 shows the results of some experiments to determine the overhead of calling methods in Objective-C and C++, compared to calling functions in C. In each test, the action to be tested has been performed 10^7 times. The time column shows the time taken, corrected for the overhead of the empty loop, for each test on the following configurations:

gcc

GNU CC 2.5.0 on a HP9000/755, compiling with -O2.

hpc

The standard C and C++ compilers on HP9000/755, compiling with +O2.

sun

GNU CC 2.3.3 on a Sun SparcStation 1.

It is clear from the table that the time it takes to invoke an Objective-C method does not differ significantly from invoking a C++ virtual method, which has semantics comparable with Objective-C methods. Of course, the direct and inlined C++ method invocation are much faster, but they are less useful in a large system where the object complexity tends to increase and dynamic binding is used a lot.

Table 2.1: Speed comparison between C++ and Objective-C

		time (ms)			relative		
Lang.	what	gcc	hpc	sun	gcc	hpc	sun
-	empty loop	210	210	1220	-	-	-
C	function	940	510	2060	1.0	1.0	1.0
C	indirect function	3410	3180	3280	3.6	6.2	1.5
C++	inline method	0	0	20	0.0	0.0	0.0
C++	method	940	420	2470	1.0	0.8	1.1
C++	virtual method	3920	3310	5830	4.1	6.4	2.8
ObjC	method	5050	n.a.	n.a.	5.3	n.a.	n.a.

3

DFGKit

In this chapter some of the design features of the DFG Manipulation Library, which was developed during the practical work being the subject of this report, are discussed. The library has been written for Data Flow Graphs as described in [vEdJS91]. By no means does the information in this chapter adequately describe how the Library actually works; it merely describes how some of the object oriented features of Objective-C are applied to the design. Information on the internals of the library can be found in appendix A.

The DFG Manipulation Library was written to be used by a VHDL to DFG compiler [Sch93c]. As such, it only provides a basic framework which however can easily be extended to include more functionality. The framework includes the following capabilities:

- Read and write the ASCII representation of ASCIS Data Flow Graphs.
- Construct, store and find graphs.
- Basic graph modification operations, like adding and removing edges, nodes and attributes.

Figure 3 shows the inheritance tree of the DFGKit classes. The top level DFG class is the DFGObject class. It represents the state and related behaviour common to all classes in DFGKit.

DFGObject inherits from a class called LispObject. This class is not part of DFGKit, but comes from Tiggr's Objective-C Library [Sch93b]. The LispObject class is used since it provides garbage collection to its subclasses.

3.1 Entities and attributes

A graph consists of nodes and edges. Each edge basically holds the same state as each other edge, like information on the edge type, it's data type and width, and source and destination nodes and ports. Each node, however, may hold information

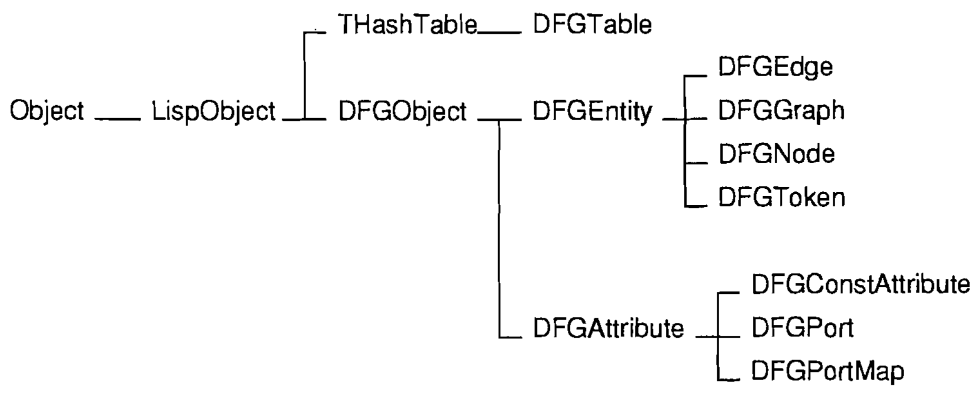

Figure 3.1: The DFGKit hierarchy.

which is only applicable to certain kinds of node, like the value of a const node, or which may vary in size, like portmaps for BMEX nodes. This leads to objects of two kinds: entities, like nodes and edges, and attributes, i.e. things that refine an entity.

When looking at the textual representation of a DFG, one thing is noteworthy about entities and attributes: Each entity is enclosed in parentheses and starts with a name describing the kind of entity, like '`(node ...)`'. Attributes, on the other hand, are not announced this way and are enclosed by the entity to which they belong.

Given this modeling in entities and attributes, it is clear that attributes are mere runtime additions to the state of an entity. This is an excellent example of dynamic inheritance through delegation.

3.2 Input and output

Input and output, are of major importance in the DFG Manipulation Library. The output features can be used to have a graph write its ASCII representation to a file. At a later point in time, such an ASCII representation can be read in order to instantiate the graphs described therein in memory.

As an example of Objective-C programming, this section will explain how the output mechanism actually works.

To output a graph, which is held by an instance of the DFGGraph class, which for instance is called `g`, the caller sends:

```
[g writeToAsciiFile: f]
```

in which f is a pointer to an open FILE. This method is only to be implemented by DFGEntity, which does nothing more than invoking

```
[self writeAscii: f]
```

followed by outputting a closing parenthesis to f. Each subclass of DFGEntity, i.e. almost all classes in the library, should implement the writeAscii: method as follows:

```
-writeAscii: (FILE *) f
{
  [super writeAscii: f];
  /* Do something specific to this class.  */
  return (self);
}
```

It is the implementation by DFGEntity which does the interesting work:

- Output to f the opening parenthesis.
- Output an identifier to identify this kind of object. This identifier is obtained through the class method typeName of the class of self.
- Output an identifier to identify this object. This is held in the name instance variable.
- Output a (type ...) clause if self has a non-nil type.
- Send a writeAscii: to each of the attributes.

In itself, this output mechanism is nothing special. It simply shows the application of dynamic binding, a concept that could easily be applied to a C++ library using virtual member functions.

There is however, one thing to note about the attributes, namely that the DFGEntity does not care about what each attribute is, as long as it responds to writeAscii:. If some subclass of DFGEntity decides add some object to its list of attributes of which the class is not known to DFGEntity at compile time, this does not matter at all. The fully dynamic binding ensures that the right method will always be invoked.

4

Conclusions

As is shown by application, a DFG Manipulation Library, written in Objective-C works. Whether this library offers any advantage to the same library written in C++ is unclear, since the Objective-C version is not complete enough yet to enable direct comparison by testing.

Furthermore, any concept that can be put in the Objective-C version of the library can also be put in the C++ version; a graph manipulation library simply is not the kind of application of a programming language where the advantages of dynamic typing become crystal clear.

Bibliography

[BW88] Hans Jürgen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, September 1988.

[Cli93] Marshall P. Cline. comp.lang.c++ frequently asked questions list (with answers, fortunately). *Usenet newsgroup comp.lang.c++*, November 1993.

[Hat93] Bob Hathaway. The comp.object faq. *Usenet newsgroup comp.object*, November 1993.

[NeX93] *NeXTSTEP Object Oriented Programming and the Objective C Language*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1993.

[Sch93a] Pieter J. Schoenmakers. Answers to frequently asked questions concerning objective-c. *Usenet newsgroup comp.lang.objective-c*, November 1993.

[Sch93b] Pieter J. Schoenmakers. Tiggr's objective-c library. Available as ftp.es.ele.tue.nl:/pub/tiggr/tobjc.tar.gz, November 1993.

[Sch93c] Pieter J. Schoenmakers. vc1, a vhdl to dfg compiler. Master's thesis, Eindhoven University of Technology, the Netherlands, Design Automation Section, December 1993.

[US91] David Ungar and Randall B. Smith. Self: The power of simplicity. *Lisp and Symbolic Computation: An International Journal*, 4(3), 1991.

[vEdJS91] J.T.J. van Eijndhoven, G.G. de Jong, and L. Stok. The ascis data flow graph: Semantics and textual format. Technical report, Eindhoven University of Technology, the Netherlands, June 1991.

[Woo93] Mark Woodruff. War stories, a report from the front line, comparing c++, eiffel and objective-c. *Usenet newsgroups comp.lang.c++, comp.lang.objective-c and comp.lang.eiffel*, May 8 1993.

A

Class Documentation

This appendix contains documentation extracted from the source files of the DFG Manipulation Library. Each section documents the methods and instance variables of a particular class. As always, see the sources for more information.

Some of the functionality of the DFGKit classes, like the application of garbage collection, list maintenance using Cons objects and the use of a Lex class to scan input files, stem from the underlying classes present in Tiggr's Objective-C Library [Sch93b]. This library also comes with the documentation extraction program that was used to generate the documentation this chapter.

In the documentation on DFGKit, it is sometimes stated that an object is of a certain class. This is, of course, an exaggeration: In all cases, the object is merely expected to behave like an instance of that class.

A.1 DFGObject

class	DFGObject
inherits	LispObject

DFGObject is the superclass of all classes in DFGKit. Most notably, it is the direct ancestor of the DFGEntity and DFGAttribute classes. The common functionality it provides is very little, i.e. `type` and `container` outlets.

Instance variables

`id type`
: The type of this object is another object, the class of which depends on the class of this instance. See the description of each subclass of DFGObject for specific typing information.

`id container`
: The container of this object, i.e. the entity to which this object belongs. This is for instance a graph for nodes and edges.

Methods

DFGObject methods

`-type`
: Return the `type` of this DFGObject.

`-setType: aType`
: Set this object's `type` to `aType`.

`-container`
: Return the `container` of this DFGObject.

`-setContainer: anEntity`
: Set this object's container to `anEntity`.

LispObject methods

`-referenceGarbage`
: After invoking `super`'s implementation, send `setGarbageReferences` to this DFGObject's `type` and `container`. Return `self`.

A.2 DFGEntity

class	DFGEntity
inherits	DFGObject: LispObject
adheres to	TableElement

DFGEntity is the superclass of all entities which are self-contained in a DFG file. Self-contained means that the entity's description starts with a '`(`', followed by an identifier describing the type of the entity (like a '`node`' or '`edge`'), followed by the semantic value of the entity and ends with the matching '`)`'.

Instance variables

`id ident`
: The identity of this entity, which is a String object.

`id attributes`
: A list of DFGAttribute objects (or objects of a subclass thereof) maintained by Cons objects.

`id extra`
: A list of anything read while parsing this entity, to which no semantic value could be assigned. The list is maintained by Cons objects and holds Cons and String objects.

Methods

DFGEntity methods

`-ident`
: Return the identity of this entity, i.e. the String object holding this entity's name.

`-setIdent: anIdent`
: Set `anIdent` to be the `ident` of this entity.

`-(const char *) name`
: Return the name of this entity. An entity's name is the `name` of its `ident`.

`-(const char *) getStringValue`
: Return the string value of this entity. An entity's string value is the string value of its `ident`.

DFGEntity attribute methods

`-addAttribute: anAttribute`

Add `anAttribute`, which is an instance of a subclass of DFGAttribute, to this entity's list of attributes. Return the attribute.

This method checks to see if this entity does not already have an attribute of that kind and sets the attributes `container` is this entity.

`-findAttributeOfType: aClass`

Return the first attribute of this entity who `isKindOf: aClass`.

LispObject methods

`-referenceGarbage`

After invoking `super's` implementation, send `setGarbageReferences` to the objects referenced by the instance variables. Return `self`.

TableElement methods

The following methods are needed for a DFGEntity to adhere to the TableElement protocol. A DFGEntity implements the Table element methods by forwarding them to its `ident`.

`-(int) compare: o`

Forwarded to `ident`.

`-(int) compareValue: (const void *) v`

Forwarded to `ident`.

`-(unsigned int) hashValue`

Forwarded to `ident`.

Archiving methods

The following methods are used for reading entities from and writing entities to an ASCII file.

`-writeToAsciiFile: (FILE *) f`

Write this entity to the ASCII file `f`. Currently, this does not output anything contained in `extra` list of things read during parsing which were not understood.

`-writeAscii: (FILE *) f`

To be implemented by subclasses to write their specific information to the ASCII file `f`. This method should first call `super`'s implementation to write out the common part. In fact, it is the implementation of `-writeAscii:` by DFGEntity that writes the first opening bracket.

`+(const char *) typeName`

Return the name of this type of entity, as it is to appear in the DFG file, like `edge` or `node`. This is only used in the DFGEntity method `-writeAscii:`; the DFG parser doesn't use it yet.

`-readFrom: lex`

Read this entity from the token stream presented by `lex` which is an instance of DFGLex. The opening bracket has been read; the current token is the type of this entity.

The implementation of this method by DFGEntity handles the identifier, if any, which follows the current token. After that, it feeds everything to `-read:startingToken:` until the next token is the token after the closing bracket matching the opening bracket for this entity.

This method is implemented by DFGEntity and should not be overridden by subclasses. Subclasses should implement `-read:startingToken:`

`-read: lex startingToken: (lex_token) token`

Read another list describing part of this entity. The opening bracket has been read. The current `token` is the first item after the opening bracket. Pass control to `super` in case this part isn't understood.

Return `self` with the current token of `lex` set to the first token after the matching closing bracket.

This is to be overridden by subclasses, which pass it to `super` in case they can't give semantic meaning to the `token`.

The implementation of `-read:startingToken:` by DFGEntity recognises the `type` token. Anything else it first tries to offer to its attributes. As a last resort, it calls `readAsciiList` and appends the list read by that function to the list pointed to by `extra`.

A.3 DFGNode

class	DFGNode
inherits	DFGEntity: DFGObject: LispObject

Instance variables

`id inEdges, outEdges`
A list of edges whose destination or source node is this node, respectively.

Inherited instance variables

`id type`
The `type` of a DFGNode normally is a DFGGraph. However, while parsing a file, the graph being the node's type might not have been defined yet. In this case, the `type` of the node is stored as the String holding the name of the type.

Methods

DFGNode methods

`-addIncomingEdge: e toPort: p`
Add the edge `e` to this node's input port `p`. Return `self`.

`-addOutgoingEdge: e toPort: p`
Add the edge `e` to this node's output port `p`. Return `self`.

`-removeIncomingEdge: e fromPort: p`
No information.

`-removeOutgoingEdge: e fromPort: p`
No information.

`-portmapForValue: (int) i nodetype: nt edgetype: et`
Return the portmap for the value `i`. If one does not exist already, create it, possibly creating the `nt` typed node in this DFGNode's container.

`nt` should be one of `dg_input` or `dg_output`, depending on whether the caller refers to a portmap on an input or output port.

This method is only applicable to BMEX nodes, i.e. nodes whose `type` is initialized to one of `dtok_branch`, `dtok_merge`, `dtok_entry` or `dtok_exit`.

DFGEntity methods

`-type`

Return the type of this node. When a DFGNode is created through `-read:startingToke` the node's type is stored as a String. When `-type` is invoked, the node attempts to resolve the name into a DFGGraph, if that has not already been done.

`-setType: aType`

Set the type of this node to `aType`, which can be a String or a DFGGraph. The node adds a port to its container if the type is `input` or `output` (either the String or the DFGGraph).

Depending on its type, the node may also add attributes to itself. For instance, if the type is `const`, the nodes adds a DFGConstAttribute to its `attributes`.

LispObject methods

`-referenceGarbage`

Send `setGarbageReferences` of the objects referenced by the instance variables. Return `self`.

Archiving methods

`+(const char *) typeName`

Return 'node'.

`-writeAscii: (FILE *) f`

After calling `super's`, write the `in-edges` and `out-edges` fields containing the edges' names.

A.4 DFGEdge

class	DFGEdge
inherits	DFGEntity: DFGObject: LispObject

Instance variables

`id srcNode`
The source DFGNode of this edge.

`id srcPort`
The source DFGPort of this edge.

`id dstNode`
The destination DFGNode of this edge.

`id dstPort`
The destination DFGPort of this edge.

`id dataType`
The data-type of this edge, which is an object responding to `name`.

`int width`
The width (in bits) of this edge. If `0`, the width is not defined.

`id varName`
The name of the variable associated with this edge.

Inherited instance variables

`id type`
The `type` of a DFGEdge is a String holding the textual representation of the edge type. It is one of `dtok_data`, `dtok_chain`, `dtok_control`, `dtok_source`, etc.

Methods

DFGEdge methods

`-getDstNode: (id *) aNode andPort: (id *) aPort`
Return in `aNode` and `aPort` the destination node and port of this edge, repsectively. Return `self`.

`-getSrcNode: (id *) aNode andPort: (id *) aPort`
Return in `aNode` and `aPort` the source node and port of this edge, repsectively. Return `self`.

`-setDstNode: aNode andPort: aPort`
Set the destination node and port of this edge to `aNode` and `aPort` repsectively. Return `self`.

`-setSrcNode: aNode andPort: aPort`
Set the source node and port of this edge to `aNode` and `aPort` repectively. Return `self`.

`-dataType`
No information.

`-(int)dataWidth`
No information.

`-variableName`
No information.

`-setDataType: t`
No information.

`-setDataWidth: (int) w`
No information.

`-setVariableName: v`
No information.

LispObject methods

`-referenceGarbage`
Send `setGarbageReferences` to the objects being referenced by this edge. Return `self`.

Archiving methods

`+(const char *) typeName`
Return 'edge'.

`-writeAscii: (FILE *) f`
After calling `super`'s implementation, write to the ASCII file `f` the `'(origin <node>)'` and `'(destination <node>)'` fields, including, if the port is not anynomous, the `'(port <portname>)'` fields. Also output typing information. Return `self`.

A.5 DFGTable

class	DFGTable
inherits	THashTable

Methods

DFGTable initialization methods

`+initialize`
: Initialize the DFGTable class. This assigns to `graphClass` the class object of DFGGraph.

`+graphClass`
: Return the class object of the class of newly to create graphs.

`+setGraphClass: aClass`
: Set the class object of the class of newly to create graphs to `aClass`.

Object methods

`-init`
: After invoking `super's`, set the delegate to the String class object.

Archiving methods

`-readAsciiFile: (const char *) fname`
: Read the dfg-view's from the file called `fname`.

`-writeToAsciiFile: (FILE *) f`
: *No information.*

Variables

`id graphClass`
: The class of newly to create graphs.

A.6 DFGGraph

class	DFGGraph
inherits	DFGEntity: DFGObject: LispObject

Instance variables

`id nodes, edges`
An AVLTree to hold this graph's nodes and one to hold its edges.

`id ports`
A list of the ports of this graph.

`int next_node, next_edge`
The number to be assigned to the next anonymous edge or node.

`BOOL associative:1, commutative:1, no_write_permission:1`
Various feature of this graph.

Methods

DFGGraph entity allocation methods

`-allocNode`
Return a newly allocated, initialized and named node. The name is obtained through `allocNodeIdent`.

`-nodeClass`
No information.

`-edgeClass`
No information.

`-portClass`
No information.

`-addNode: aNode`
Add the node `aNode` to this graph, calling the node's `setContainer:`. Return `aNode`.

`-addEdge: anEdge`
Add the edge `anEdge` to this graph, calling the edge's `setContainer:`. Return `anEdge`.

`-addPort: aPort`
Add the port `aPort` to this graph's `ports`. Return `aPort`.

`-nodes`
Return the AVLTree holding the `nodes`.

`-edges`
Return the AVLTree holding the `edges`.

`-ports`
Return the list of the `ports`.

`-allocNodeIdent`
Return a String holding a name usable as a node identifier which is unique within this graph.

`-allocEdgeIdent`
Return a String holding a name usable as a edge identifier which is unique within this graph.

`-addMatchingEdgeFromNode: ns port: ps toNode: nd edgeType: et`
Attempt to create an edge originating at the port `ps` of node `ns` to some port of node `nd`, suggesting edge-type `et`. Returns the newly allocated edge, or nil upon failure. The new edge is anonymous, connected, and added to this graph's list of edges.

`-addMatchingEdgeFromNode: ns port: ps toNode: nd argno: (int) n edgeType: et`
Attempt to create an edge originating at the port `ps` of node `ns` to some port of node `nd`, suggesting argument number `n` and edge-type `et`. Returns the newly allocated edge, or nil upon failure. The new edge is anonymous, connected, and added to this graph's list of edges.

`-addEdge: et fromNode: ns andPort: ps toNode: nd andPort: pd`
If the edgetype `et` is not prescribed, use the edge-type of the source port. If that too is nil, use `dtok_data`. Return the newly created edge.

`-portForArgno: (int) n`
Return the first port with the argument number `n`, or `nil` if `n == -1`.

`-portForArgno: (int) n type: io`
Return the first port with argument number `n` whose container type is `io`. Normally, `io` is one of `dg_input` or `dg_output`.

`-portForName: name type: nt edgetype: et`
Return the port named `name`, creating a new `nt` typed node if the port does not already exist. `nt` should be one of `dg_input` and `dg_output`. XXX Does not do anything with `et` yet.

`-findPortTyped: pt edgetype: et`
Find a port for the edgetype `et` and the container type `pt`. Normally, `pt` is one of `dg_input` or `dg_output` and `et` is one of `dtok_data`, etc. If `et` is

not nil and an exact match can not be made, the first port with a nil et is returned.

LispObject methods

-referenceGarbage
    Send setGarbageReferences to the objects referenced by this DFGGraph. Return self.

Object methods

-init
    No information.

Archiving methods

+(const char *) typeName
    Return 'graph'.

-writeAscii: (FILE *) f
    After invoking super's implementation, send writeToAsciiFile: to this graphs nodes and edges. Return self.

-read: lex startingToken: (lex_token) token
    If the token is dtok_node, allocate a new node, set its container to the current graph's self, have the node read itself and, finally, add it to this graph and return self. The same is true for edges if the current token is dtok_edge. Otherwise return the return value of super's implementation.

A.7 DFGToken

class	DFGToken
inherits	DFGEntity: DFGObject: LispObject

Instance variables

`id node`
The source DFGNode of this token.

`id port`
The source DFGPort of this token.

`int width`
The width (in bits) of the data in this token. If `0`, the width is not defined.

`id varName`
The name of the variable associated with this token, or `nil` if there's no such association.

Inherited instance variables

`id type`
The `type` of a DFGToken is an Object identifying the data type held by this token. It responds to `name` in order to release its textual representation.

Methods

DFGToken methods

`-node`
No information.

`-port`
No information.

`-(int)dataWidth`
No information.

`-variableName`
No information.

`-setNode: n`
No information.

`-setPort: p`
No information.

`-setDataWidth: (int) w`
No information.

`-setVariableName: v`
No information.

DFGToken methods

`-initNode: n port: p`
No information.

LispObject methods

`-referenceGarbage`
Send `setGarbageReferences` to the objects being referenced by this token. Return `self`.

A.8 DFGAttribute

class	DFGAttribute
inherits	DFGObject: LispObject

DFGAttribute is the superclass of all attributes which a DFGEntity can have. An attribute contains extra information on an entity, which is so large that putting this information in every entity would be a waste of memory.

An DFGAttribute is very different from a DFGEntity with respect to the way it is created during parsing a DFG file: A DFGEntity is created by its container upon encountering '(<entity>', where '<entity>' is an identifier describing which self-contained entity is defined by the list between the opening bracket and the matching closing bracket. On the other hand, a DFGAttribute is created by its container upon encountering tokens which add semantic meaning to the entity, in order to hold that semantic meaning.

The DFGAttribute basically is just a placeholder for subclasses, which are real attributes: it does not have any instance variables and its methods basically do nothing.

Methods

Archiving methods

`-read: lex startingToken: (lex_token) token`

Attempt to read another semantic item, which this attribute's entity is trying to feed to one of its attributes.

The `'('` has been read. `token` is the current token of `lex`; it is the first item after the `'('`.

If the `token` makes sense, return `self` with the current token of `lex` set to the first token after the matching `')'`. Otherwise, pass it to `super`. The implementation by DFGAttribute of '-read:startingToken:' simply returns `nil` indicating that the token did not make sense to this attribute.

`-writeToAsciiFile: (FILE *) f`

Return `[self writeAscii: f]`.

`-writeAscii: (FILE *) f`

To be implemented by subclasses to write their specific information to the ASCII file `f`. This method should first let `super`'s implementation do its work. The implementation by DFGAttribute returns `self`. -

A.9 DFGConstAttribute

class	DFGConstAttribute
inherits	DFGAttribute: DFGObject: LispObject

The DFGConstAttribute class is a DFGAttribute that can hold constant numeric (integer or floating point) values.

Instance variables

`id value`
  The object actually holding the value.

Methods

DFGConstAttribute methods

`-setValue: o`
  Set this attribute's value to be the object `o`.

`-setIntValue: (int) i`
  Have the receiving object hold the integer value `i`. Return `self`.

`-setDoubleValue: (double) d`
  Have the receiving object hold the floating point value d. Return `self`.

`-(int) getIntValue`
  Return the integer value this object is holding.

`-(double) getDoubleValue`
  Return the floating point value held by this DFGConstAttribute.

LispObject methods

`-referenceGarbage`
  Send `setGarbageReferences` to the `value` of this DFGConstAttribute. Return `self`.

Archiving methods

`-read: lex startingToken: (lex_token) token`

Attempt to read `'(const-value <integer>)'` and return `self`. Otherwise, return `super`'s implementation return value.

`-writeAscii: (FILE *) f`

Write `'(const-value <value>)'` to `f` after hvaing called `super`'s implementation. Return `self`.

A.10 DFGPort

class	DFGPort
inherits	DFGAttribute: DFGObject: LispObject

Instance variables

`int argno`
: The argument number of this port, or -1 if it wasn't set.

`BOOL anonymous:1`
: If `TRUE`, it is allowed to refer to this port without explicitly mentioning its name.

Inherited instance variables

`id type`
: The `type` of a port indicates, using a String, the default edge-type of edges connected to that port.

Methods

DFGPort methods

`-(const char *) name`
: Return the name of the receiving object's `container`.

`-(BOOL) anonymous`
: Return whether this port may be used anonymously.

`-setAnonymous: (BOOL) b`
: Set the anonymicity of this DFGPort to `b`. Return `self`.

`-(int) argno`
: Return the `argno` of the variable in the function for which this port's graph is a representation.

`-setArgno: (int) n`
: Set the argument number of this port to `n`. Return `self`.

Object methods

`-init`

After calling `super's` implementation, set the `argno` to −1. All other fields have already been initialized to 0, which are the correct values. Return `self`.

Archiving methods

`+(const char *) typeName`

Return '`port`'.

`-read: lex startingToken: (lex_token) token`

Attempt to read '`(anonymous <boolean>)`', '`(argno <integer>)`' or '`(edgetype <edgetype>)`'and return `self`. Otherwise, return `super's` implementation return value.

`-writeAscii: (FILE *) f`

After calling `super's` implementation, write, if applicable, the '`anonymous`', '`argno`' and '`edge-type`' fields to `f` and return `self`.

A.11 DFGPortmap

class	DFGPortmap
inherits	DFGAttribute: DFGObject: LispObject

Instance variables

`id port`

No information.

`int *values, num`

No information.

Methods

`-setPort: aPort`

No information.

`-port`

No information.

`-addValue: (int) v`

No information.

`-(unsigned int) count`

No information.

`-(int) valueAt: (int) i`

No information.

`-(int) findValue: (int) v`

No information.

`-read: lex startingToken: (lex_token) token`

No information.

`-portmapWriteAscii: (FILE *) f`

No information.