OOPAL: Integrating Array Programming in Object-Oriented Programming

Philippe Mougin
pmougin@acm.org

Stéphane Ducasse
Software Composition Group
University Of Bern
Bern, Switzerland
ducasse@iam.unibe.ch

ABSTRACT

Array programming shines in its ability to express computations at a high-level of abstraction, allowing one to manipulate and query whole *sets* of data at *once*. This paper presents the OOPAL model that enhances object-oriented programming with array programming features. The goal of OOPAL is to determine a minimum set of modifications that must be made to the traditional object model in order to take advantage of the possibilities of array programming. It is based on a minimal extension of method invocation and the definition of a kernel of methods implementing fundamental array programming operations. The OOPAL model presents a generalization of traditional message passing in the sense that a message can be send to an entire set of objects. The model is validated in F-SCRIPT, a new scripting language.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Object-oriented languages, APL*; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Language, Design, Performance

Keywords

F-Script, Message Pattern, Array Programming, Smalltalk, High-Order Messages, High-Level Language

1. THE PROBLEM

While object-oriented programming offers high-level tools for data *modeling* (abstract data type, polymorphism, inheritance), the same is not true for data *manipulation*. The basic operation provided for object manipulation is message sending, a fundamental operation as it supports polymorphism and encapsulation, but which remains a very fine-grained low-level operation. Indeed object-oriented programming does not offer a high-level model for manipulating whole sets of data (*i.e.*, collections of objects). In high-level programming models like array programming or relational algebra, complex expressions manipulating entire sets of data are easy to design and are expressed in an extremely compact manner, without requiring the explicit use of loops, tests, or navigation instructions in a data graph. The same expressions would require several tens or hundreds of lines of code in conventional object-oriented programming languages.

Our aim is to provide a new approach which offers higher-level capacities for data manipulation within the scope of object-oriented programming. For this purpose, we enrich object-oriented programming with concepts taken from array programming in a model called OOPAL (which stands for Object-Oriented Programming and Array programming Language integration). This paper presents OOPAL and its implementation in F-SCRIPT, an object-oriented scripting language using a Smalltalk syntax. This article makes the following contributions: it identifies design principles for successful integration between OOP and array programming, defines the OOPAL model that enables this integration, and shows how it is validated through implementation of F-SCRIPT.

We start by an overview of array programming (Section 2). An example in traditional object-oriented language is compared with its equivalent in F-SCRIPT (Section 3). An analysis of the requirements for obtaining a successful integration is then presented in terms of design principles (Section 4). Based on these principles, OOPAL is described as three components: message patterns (Section 5), mapping between basic array programming and OOP concepts (Section 6), and array programming operations (Section 7).

2. ARRAY PROGRAMMING OVERVIEW

Array programming is the result of a mathematical notation invented by Kenneth Iverson in the 1950s. The PAT system (Personalized Array Translator) was the first computing environment to implement the Iverson notation, followed by APL, which was developed by IBM in the 1960s and earned Iverson the Turing Award [28, 6]. Herb Hellerman, the creator of PAT, explains the choice of an array language in [23]:

> Major advances in programming language are desirable to best provide a personalized service for creative man-machine interaction. A most promising direction for improvement is to better respond to the fact that most problems presented for computer solution deal with arrays of operands rather than isolated operands. Although modern languages include the ability to access array elements by indexing, the Iverson language recognizes, in its basic structure, the ability to specify entire arrays as operands. This not only reduces the number of program characters which must be speci-

OOPSLA'03, October 26–30, 2003, Anaheim, California, USA.

fied to perform most program functions, it also reduces the amount of loop control and other explicit program logic required of the user. The language includes selection operations, generation of test and argument arrays, and other macro-operations in a simple consistent manner.

Array programming is used in numerous fields and in particular in science and finance. Some of its concepts are found in some widely used software such as MatLab, Mathematica and IDL[42]. APL continues to be developed and has inspired numerous other languages such as ZPL [13], Nial [39, 34], Fortran 90 and HPF [53], K [47], J [48, 26, 27] and PDL [20].

2.1 Principles

Array programming has two key characteristics:

- Operations can be directly applied to entire arrays of values.
- A set of special functions and operators provides powerful means of data manipulation and allows complex data manipulation processes to be expressed concisely.

The fundamental principle behind array programming is that operations are directly applied to entire arrays of values, without the need for explicit loops. For example, if X and Y are two arrays of numbers, X+Y returns a new array which contains the sum, elementwise, of X and Y as shown by Figure 2. It is also possible to combine scalars and arrays in the same expression. For example, X*2 returns an array which contains the result of multiplying each element in X by two. In an array programming language such as Fortran 90, the expression Z = W+X*SIN(Y) is legal, not only when W, X, Y, and Z are scalars, but also when they are arrays.

2.2 Array Programming Building Blocks

More precisely, array programming is built around scalars, multi-dimensional arrays, functions, and operators.

Scalars. Conventional array programming languages are oriented towards numeric computing. They offer a certain number of basic data types, including numbers and characters, and sometimes offer complex numbers, dates, and Booleans.

Multi-Dimensional Arrays. Traditionally, array programming supports multi-dimensional arrays and lets one specify the dimension onto which operations are carried out. For instance, in APL, one can index the operation symbol with a number specifying the dimension using [and]. Thus, in the expression X/[Z]Y, the array Y is compressed along its Zth dimension using the Boolean array X.

Functions and Operators. Furthermore, array programming features a set of powerful functions and operators which enable data manipulation to be expressed in a concise manner. As pointed out by the ACM SIGAPL, "The APL primitives express broad ideas of data manipulation. These rich and powerful primitives can be strung together to perform in one line what would require pages in other programming languages". For instance, these primitives allow one to: select some elements in an array, cumulatively apply a function to the elements of an array, transpose the elements of an array, slice an array, reshape an array, or combine array elements with generalized outer and inner product operations. These array programming operations are in synergy with the automatic processing of arrays and offer great power for data manipulation. For example, *compression* may be used to select elements in an array which meet a particular criterion. Thus, in APL the expression (X < 60) / X selects elements from X which are less than 60.

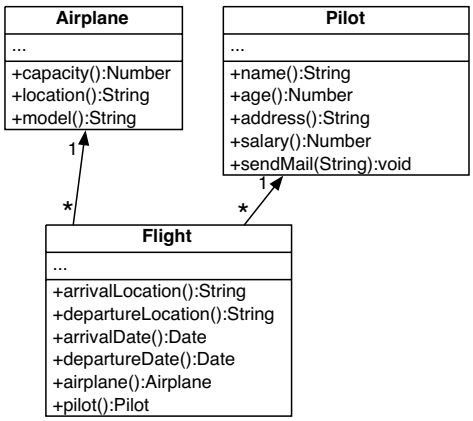

Figure 1: A simple domain: airplanes, flights and pilots.

3. AN EXAMPLE

To illustrate our point we use the object model presented in Figure 1 in several examples throughout the paper. It presents the minimal model of an airplane company and represents flights, airplanes, pilots, and their relationships (an airplane and a pilot are associated with each flight). We are only interested in manipulating objects via their behavioral interface. Thus, our UML schema does not specify the instance variables but only the methods. We define three collections of objects named A, F, and P which group together the references to all instances of airplanes (A), flights (F), and pilots (P) in our airplane company's fleet.

Suppose that we want to print the names of all the pilots, ranked by salary in increasing order, in charge of a flight to Paris on a B747 airplane. Using Java or other mainstream object-oriented languages we would have to write this kind of code[1].

Java.

```
TreeSet pilots = new TreeSet(new Comparator()
{
   public int compare(Object o1, Object o2)
   {
      int salary1 = ((Pilot)o1).salary();
      int salary2 = ((Pilot)o2).salary();
      if (salary1 < salary2)
         return -1;
      else if (salary1 == salary2)
         return 0;
      else
         return 1;
   }
});
Iterator flightIterator = F.iterator();
while (flightIterator.hasNext())
{
   Flight currentFlight = (Flight)flightIterator.next();
   if (currentFlight.arrivalLocation().equals("PARIS")
   && currentFlight.airplane().model().equals("B747"))
   {
      pilots.add(currentFlight.pilot());
   }
}
Iterator pilotIterator = pilots.iterator();
```

[1]By using a TreeSet we can both sort the selected pilots by salary in increasing order and avoid duplicates.

```
while (pilotIterator.hasNext())
{
   System.out.println(((Pilot)pilotIterator.next()).name());
}
```

Our solution, OOPAL, is based on the unification of array programming and OOP. The main idea is that this unification lets one develop code using the object interfaces and still use the power of array programming. The unification is realized by the introduction of *message patterns* and array programming operations (See Section 5). As the OOPAL model is supported by F-SCRIPT, a new Smalltalk dialect, we use it as a notation to express the examples.

With F-SCRIPT, the same operation on the same object model is expressed as follows:

F-SCRIPT.
```
pilots := (F at:F arrivalLocation = 'PARIS'
              & (F airplane model = 'B747')) pilot distinct.
sys log:(pilots name at:pilots salary sort)
```

The expression F arrivalLocation = 'PARIS' returns an array of Booleans that indicates, for each flight, whether or not the arrival location is 'PARIS'. Such an array combined with the other expression F airplane model = 'B747' is then used by the at: method to select the corresponding flights.

4. CHALLENGES OF SUCCESSFUL INTEGRATION

Based on the analysis of existing object-oriented languages and array languages, this section presents the challenges that need to be resolved to integrate array programming in OOP and defines a list of design principles that such an integration should satisfy.

4.1 Design Principles

The OOPAL model is the result of a simple analysis: since message passing is the fundamental operation in object-oriented programming, an array programming operation such as the addition of two numerical arrays must be conveyed as the generation of a certain number of messages over the array elements. For example, suppose X and Y are two arrays of the same size whose elements are instances of the Number class, with X = {1, 2, 3, 4} and Y = {10, 20, 30, 40}. Then X+Y must result in the generation of four messages: 1+10, 2+20, 3+30, 4+40 (Figure 2).

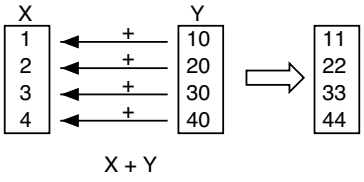

Figure 2: X+Y results in the array {11, 22, 33, 44}.

Supporting All Types of Data. Traditionally, array programming supports the manipulation of numbers and characters. For integration with OOP, we want to be able to manipulate any object with the same ease.

Therefore, the solution should use objects as atomic data elements, and should not be limited to numbers and characters alone.

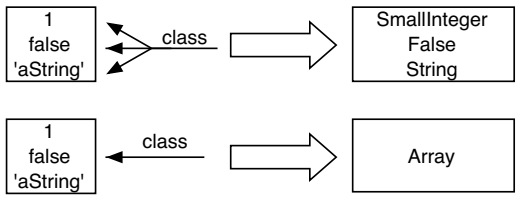

Figure 3: The need to differentiate messages sent to an array or to its elements.

Minimal Extension to OOP. The integration of array programming techniques with object technology can only be broadly accepted if array programming is a new conceptual tool that OOP developers can use when appropriate. The foundation of object-oriented programming should therefore not be altered. Ideally, it would require no change or extension of existing languages.

The solution must minimize the extensions required to conventional OOP, while featuring the main advantages of array programming.

Extensibility. A common approach for supporting array programming features is to implement operations in a way that makes them able to automatically process arrays element-by-element. This has been adapted to some object-oriented languages. In Squeak [25], for instance, some arithmetic methods are implemented in that way, making it possible to write X+Y to add whole arrays. However, in an object-oriented context, it is hard to generalize this solution. Indeed, generalizing this approach to an existing object-oriented system would require reformulating and re-implementing each existing method to make it able to function in *array mode*. Furthermore, it would require programming each new method in that way. We want the OOPAL array programming model to be universally applicable (*i.e.*, not restricted to a set of predefined methods) and we don't want it to put any burden on class developers.

The proposed solution should be generic and not based on a predefined identification of the methods. It should work for all methods, without requiring any special support from the methods themselves.

Supporting Encapsulation. Encapsulation - manipulating objects using their methods - is consubstantial with the notion of objects and supports desirable characteristics such as polymorphism, implementation hiding, extensibility, integrity enforcement [45]. The failure to support encapsulation leads to a number of well-known problems [32].

The solution must therefore support encapsulation and consider that interaction with objects is carried out by sending messages.

No Additional Constraints on Objects. Sometimes, solutions for integrating object technology with other paradigms or the use of object-oriented frameworks place restrictions on the object-oriented programming model. Such practices lead to upfront design decisions that force developers to consider how the objects they are designing will be manipulated. Within the scope of the OOPAL model, we believe that such constraints are undesirable.

Therefore, the solution should not place any constraint on objects. Developer should not need to know in advance whether their objects will be manipulated using the OOPAL *model.*

Handling Array Specific Messages. Since arrays are also objects, it should be possible to send messages to the arrays themselves and not their elements. As shown by Figure 3, we may want to access the elements of an array or invoke methods of the array itself.

The solution must provide a means of differentiating the operations on arrays from operations on the contents of arrays.

Nested Arrays. An operation can apply to different nested levels as shown by Figure 4. You may wish to go to the maximum nesting level, stop at the first level, or *go down* to an arbitrary intermediate nested level.

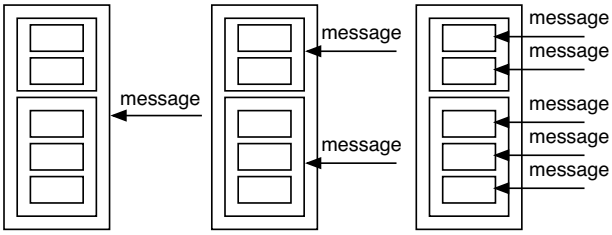

Figure 4: The need to specify the level to which the message applies.

The solution must support the specification of the nested level to be reached (this is true for the receiver as well as for arguments).

Various Combinations. Until now we have imagined the implementation of operations that associate array elements one by one. However, some other useful combinations are possible. For example, we may want to execute an outer product as in APL, *i.e.*, combine an array element with each element of another array as shown by Figure 6. The number of possible combinations becomes even larger if we consider messages with any given number of arguments. Indeed, these messages can bring any number of arrays and scalars into play as shown in Figure 5.

Therefore the solution must cover a large number of useful combinations.

4.2 The OOPAL Model in a Nutshell

As we will present in Section 5, the OOPAL model is based on *message patterns* that support the manipulation of object collections in an array programming-like fashion. Message patterns support encapsulation, extensibility, a minimal extension of the object model, and the handling of array specific messages and nested arrays. They also avoid the addition of constraints on object interfaces. Therefore *any* kind of object and *any* kind of method can be manipulated in an array programming-like fashion. The model is completed with a set of specific array programming operations that have been adapted to object-oriented programming.

5. MESSAGE PATTERNS

Now, we shall present the core aspect of OOPAL, *message patterns*[2]. As we have mentioned, message patterns are at the heart of the minimal extension of the traditional object model, which supports the design principles identified in Section 4.

[2]"Message pattern" is sometimes used in Smalltalk as a synonym for selector with method argument. This differs from our usage in this paper.

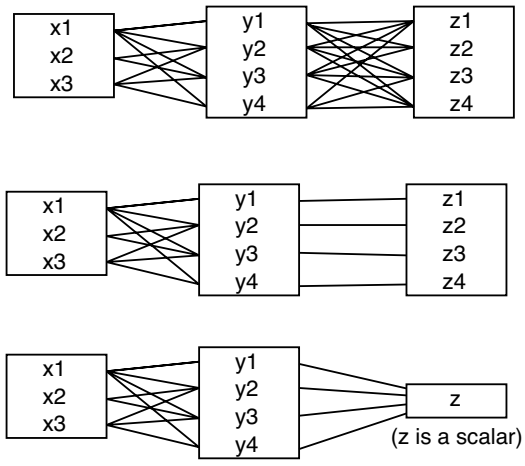

Figure 5: Array elements and scalars may be combined in complex ways.

5.1 Extended Message Passing

As explained, array programming enables operations to be applied to entire arrays without requiring the explicit use of a traditional loop control structure. OOPAL message patterns extend the traditional message passing operation in a backward compatible manner to support messages to be sent to collection of objects. Message patterns allow for the sending not only of a simple message, but of a complex group of messages. Traditional message passing then conceptually becomes a specific case of message pattern. Note that as regards speed of execution, normal method invocation is not affected by message patterns.

In this paper we present the principal elements of message patterns. A complete description can be found in the F-SCRIPT User's Manual [36].

Message pattern notation makes it possible to:

- Specify for each array involved in a message pattern (*i.e.*, receiver and arguments) whether a loop should iterate over the elements of this array. Moreover, the nesting level of this loop in relation to other loops in the message pattern can be specified.
- Specify a different message pattern for each level of array nesting, should nested arrays be used.
- Use *implicit message pattern* which makes notation easier.

We also propose an abstract notation that makes it possible to express the structural properties of message patterns, regardless of their actual message selectors (*i.e.*, method names) and arguments. These structural properties are called patterns. For example, the notion of an outer product corresponds to a particular pattern.

5.2 Simple Message Patterns

The message pattern notation involves indicating iterations in the message expression itself. When the @ symbol is placed after the receiver and/or just before an argument, it means that a loop iterating over the elements of such designated array(s) is executed to generate message sends.

The two following rules using a Smalltalk syntax[3] express it (these rules can be combined, as we show later):

- rec @max:arg means that all the elements of rec receive the message max: with the argument arg. {1, 2, 3} @max:2 generates three message sends (*i.e.*, 1 max:2, 2 max:2, 3 max:2) and returns {2, 2, 3}.
- rec max:@ arg means that rec receives the message max: for each elements of arg. 2 max:@ {1, 2, 3} generates three message sends (*i.e.*, 2 max:1, 2 max:2, 2 max:3) and returns {2, 2, 3}.

For example, the expression F @airplane sends the message airplane to all the elements of F and returns the resulting array (*i.e.*, an array of airplane objects, having the same size as F, where the airplane at index i in this array is the airplane associated with the flight at index i in F). One can note that the @ form is very similar to the EACH operator in APL.

Both receiver and arguments can be iterated upon as shown by the following example: {1, 2, 3} @+@ {10, 20, 30} returns the array {11, 22, 33}

As shown by the second rule, a message pattern expression does not require the receiver to be an array. In this case, the message pattern will only lift over the arguments designated by an @ sign. For instance, suppose that D is a dictionary providing lookup with the method objectForKey:. D contains associations between country names and capital city names. To get the capital city of France, we write D objectForKey:'France', which returns 'Paris'. Thanks to message patterns, we can also provide a whole set of keys without requiring any special support from the objectForKey: method: D objectForKey:@{'France', 'Norway', 'Canada', 'Japan' } returns {'Paris', 'Oslo', 'Ottawa', 'Tokyo'}.

The messages resulting from the execution of a message pattern are sent sequentially. There are no parallel or asynchronous semantics attached to OOPAL message patterns[4]. Arrays are iterated from the start of the array towards the end. The method invocation semantic is not altered by OOPAL and is defined by the object-oriented language at use.

Composition. A message pattern has the same precedence level as conventional message sends and can be composed in a similar manner. For example, the expression F @airplane @model generates the airplane array described in a previous example, and then sends the model message to each element in this array, which returns a new array giving the airplane model corresponding to each flight. The expression F @departureDate @> NSDate now returns a Boolean array which indicates, for each flight, whether it will take place after the current date [5].

[3] In Smalltalk, we invoke the method max: with parameter b on object a with the expression a max:b – this is the Smalltalk equivalent of writing a.max(b) in Java. The method's name is "max:" and is called the selector. Unary selectors (*i.e.*, selectors with no argument) have a higher precedence than binary selectors (*i.e.*, selectors using symbols like +, -, < etc.), which have themselves a higher precedence than keyword selectors (*i.e.*, selectors that includes one or more colons, such as max: or at:put:). Thus the expression a max:b+c abs will be evaluated in the same way as a max:(b+(c abs)).

[4] This is to cope with the *Minimal Extension to OOP* requirement expressed in section 4.1, as introducing parallel execution semantics would clearly not contribute to a minimal extension. That said, the message pattern idea and notation certainly has good potential to be associated with parallel programming semantics. This subject is, however, outside the scope of this article.

[5] NSDate is the class representing the dates in F-SCRIPT and now is a class method returning the current date.The NSDate class implements conventional comparison methods.

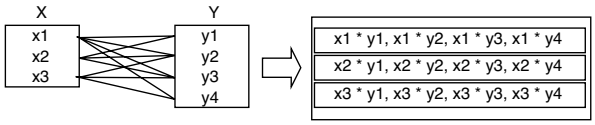

Figure 6: Outer product.

Advanced Combinations. Thus far, we have seen that by using patterns on some arrays we generate messages that use the first element of each array, then the second, and so on. But we can also combine array elements in other ways. For example, suppose we want to get the outer product of X and Y, using the * method (Figure 6). To do this, we have to specify that we want a loop on X and an *inner* loop on Y. We therefore use a number after the @ symbol to state the inner level of each loop. The outer product as shown in Figure 6 is then expressed as X @1*@2 Y.

For instance, if X is {1, 2, 3} and Y is {10, 20, 30}, then X @1*@2 Y returns {{10, 20, 30},{20, 40, 60}, {30, 60, 90}}.

If Z is {2, 0, 4} then X @1 between:@2 Y and:@3 Z returns {{{false, true, false}, {false, true, false}, {false, true, false}}, {{true, true, false}, {true, true, false}, {true, true, false}}, {{true, true, false}, {true, true, false}, {true, true, false}}}.

X @1 between:@2 Y and:@1 Z returns {{false, false, false}, {true, true, true}, {false, false, false}}

5.3 Recursive Message Patterns

Simple message patterns are the most commonly used. However they are not sufficient when dealing with nested arrays. For example, let X be the array {{1, 'string', {3}}, {5}}. X has two elements. The first element is an array containing a number, a string, and another array, and the second element is an array containing a number. The expression X class returns the class Array. The expression X @class returns {Array, Array} as the message class is sent to the array elements. To get the class of the elements of each element we use a recursive message pattern as follows: X @@ class which returns {{Number, String, Array}, {Number}}. In a recursive message pattern, a pattern is applied to another pattern instead of just to a message. Recursive message patterns have multiple levels. At each level is a pattern that applies to the pattern found at the next level. The pattern at the last level applies to a simple message. In the X @@class example, the first level of the message pattern is the leftmost @. It indicates that the rest of the message pattern, here @ class, must be applied to each element in array X. Each level in a recursive message pattern corresponds to a nesting level in the arrays being manipulated.

Recursive message patterns can easily become very complex, but such complex message patterns are rarely used.

5.4 Implicit Message Pattern Notation

The notation introduced previously supports the definition of message patterns and is fully executable in F-SCRIPT. However, this explicit notation is still too complex compared with conventional

array languages. In APL we write X+Y to add the arrays X and Y, with the notation presented above we must write X @+@ Y.

To solve this problem, we introduce an *implicit* notation for message patterns, which in most cases eliminates the explicit use of @. Going back to our example, we can now write X + Y like in APL.

This approach requires the language in which it is used to be able to recognize the specification of a message pattern even if the message pattern is not denoted by any particular signs. In F-SCRIPT, we use the following rule to support implicit message patterns: when a message is sent to an array, if the method invoked is not defined by the class Array, the message is sent to the receiver elements. Moreover, if certain arguments of this message are also arrays F-SCRIPT takes the elements in these arrays one at a time.

When X, Y, and Z are arrays of numbers, this rule makes it possible to write expressions such as: X+Y, X*2, X cos, X between:Y and:Z, and X between:10 and:Z. These expressions follow the array semantics *i.e.*, they operate on the array elements. The rule also makes it possible to use the implicit notation for recursive message patterns. For instance {{1, 2, 3}, {10, 3.14}} * 2 is equivalent to {{1, 2, 3}, {10, 3.14}} @@* 2 and evaluates to {{2, 4, 6},{20, 6.28}}.

Thanks to implicit notation we can simplify the examples given in the previous sections. Thus, we can write F airplane instead of F @airplane and F airplane model instead of F @airplane @model. However, implicit notation cannot always be used, especially when the receiver is not an array but arguments should be iterated upon. For instance, the expression 2 * X where X is an array of numbers will not work. In this case, an explicit message pattern is required: 2 *@ X.

Note that the proposed rule can be implemented in the message-sending routine with nearly no performance loss.

5.5 Abstract Notation

When reasoning about message patterns, we may not be interested by the actual message selector and the actual arguments used in the expression. We often want to think about the structural part of a message pattern expression. We extract this structural information, which we call a pattern, and represent it as follow:

- The pattern of the message pattern expression F @arrivalDate is @.
- The pattern of the message pattern expression {1, 2, 3} @+@ {10, 20, 30} is @:@.
- The pattern of the message pattern expression 2 max:@{0, 4, 8} is :@.

Given an explicit message pattern expression in F-SCRIPT, one can produce the associated abstract notation by applying the following transformations:

- Remove the sub-expressions representing the receiver and the arguments.
- Replace binary selectors (*e.g.*, +, -, etc.) by a colon.
- Remove all the characters from the selector except the colon characters.

Some patterns are frequently used and can be given standard, well-known names. For example, @1:@2 is called *outer product*.

6. BUILDING BLOCKS

The OOPAL model would not be complete without implementing the basic elements of array programming. Now we present how we map them into an object-oriented programming model.

6.1 Scalars

The scalar data types (*e.g.*, Booleans, numbers) manipulated by conventional array programming languages corresponds in our model to classes. Data is no longer limited to certain types but may be of any class. This is possible because, in OOPAL, array programming facilities are universal and not linked to any particular class.

6.2 Multi-dimensional Arrays

Array programming allows data to be grouped together in *multi-dimensional* arrays which vary in size, are heterogeneous, and may themselves contain arrays.

In object-oriented languages the closest data-type is *ordered collection* (*e.g.*, List in Java, OrderedCollection in Smalltalk). OOPAL uses the extensibility of OOP to define the main array-programming operations as dedicated methods on collection classes. In this article, we refer to these enriched collections as "arrays".

Conventional object collections are one-dimensional and do not offer direct support for the multi-dimensional manipulations which are so dear to array programming. Nonetheless, we believe that it is preferable in our context to avoid introducing the notion of multi-dimensional object collections for three reasons.

- First, it would involve a substantial addition leading to numerous complexities and this would contradict the minimal extension principle.
- Second, while multi-dimensional matrix manipulation may be a common occurrence in mathematical processes it is less useful in our context which looks at the manipulation of any type of objects.
- Third, we are not abandoning advanced support for operations requiring multi-level object nesting. Collection nesting is still possible, as nothing prevents a collection from containing collections. In OOPAL, multi-dimensional arrays are simulated using nested arrays. The notion of recursive message patterns allows array programming techniques to be used on nested arrays regardless of the depth of nesting, and the transposition operation (see section 7) caters for explicit permutation of dimensions.

6.3 Functions and Operators

Array programming languages offer a certain number of operations, called *functions*, which may be applied to scalar data and/or to arrays. Functions may be provided by the language such as basic mathematical operations or array-specific functions like compression. They may also be provided by the developer or via external libraries. In OOP, functions map naturally to methods.

Array programming also uses the notion of *operators* (see [29]). An operator can be described as a function which applies to other functions to produce a function. Array programming is widely based on operators for data manipulation. For example, the reduction operator enables any function with two arguments to be cumulatively applied to all the elements in an array. As with functions, primitive operators exist and it is possible to define new operators.

Object-oriented programming does not have any direct equivalent to the operator concept but the main advantages of operators can be obtained by implementing methods which use *operations* as arguments. Depending on the language, one can use constructions such as blocks (lambda functions) or selectors in Smalltalk, anonymous methods in C#, selectors in Objective-C, or lambda functions in Python, CLOS. For example, an equivalent to a reduction operator (in a slightly modified form) already exists in Smalltalk with the inject:into: method, which takes a block as an argument.

7. ARRAY PROGRAMMING OPERATIONS BY EXAMPLE

The notion of message pattern alone is not sufficient to fully integrate array programming in object-oriented programming. Message patterns become powerful when they are associated with specific array programming operations. Note that these operations are implemented as simple methods. No new syntactic notation is necessary. These operations are generic and are meant to be provided by languages supporting the OOPAL model. They do not require support from the developer. In particular, these operations enable:

- Easy navigation in the object graph,
- Concise and readable expression of the selection of objects according to arbitrarily complex criteria,
- Sophisticated data analysis, and
- Concise and readable expression of complex object manipulations.

As we illustrate now, using message patterns and these operations makes it possible to capitalize on the power of array programming principles in an object-oriented context.

7.1 Six Selected Operations

Here, we present six examples of array specific operations, as implemented in F-SCRIPT, but there are many others (see [36, 37]).

Compression. Compression selects certain elements of an array. The result of compression is a new array which contains the selected elements. Compression is an operation which requires two operands: the array from which we want to make a selection and a *Boolean array* of the same size. An element is selected if the corresponding Boolean (*i.e.*, at the same index) holds true. For example, suppose that our array P contains eight Pilot objects. If we want to select elements at index 0, 1, and 5, we can apply compression as follows[6]: P at:{true, true, false, false, false, true, false, false}.

Compression requires a Boolean array which specifies the selection that has to be made. Thanks to message patterns, such arrays are very easy to generate. For example, P salary $>$ 3000, generates a Boolean array which indicates, for each pilot, whether his/her salary is greater than 3000. The expression is evaluated as follows: the salary message is sent to each element in P (this message pattern is denoted implicitly), which produces an array of numbers. The message $>$ 3000 is then sent to each element in the array of numbers (here again, in accordance with the implicit notation of message patterns) which produces the desired Boolean array. We can then use this Boolean array to compress the P array and thus select only pilots whose salary is greater than 3000: P at: P salary $>$ 3000.

Message patterns enable complex Boolean conditions to be expressed naturally. For example, (P at:(P salary $>$ 3000) & (P address = 'PARIS')) sendMail:'Dear Pilot, etc. etc. etc.' selects the pilots whose salary is greater than 3000 and who live in Paris, and sends them an e-mail.

Reduction. Reducing an array consists in cumulatively evaluating a custom operation on the elements of an array. In F-SCRIPT, reduction is implemented as the method \ of the Array class. For example, you add up the elements of an array with the expression: {1, 2, 3, 4} \ #+ which returns 10. The result is computed as if you had entered: 1 + 2 + 3 + 4. Reduction may be used with any operation (method or block) which takes two operands and returns an object. Some types of reduction are used very often. For example:

- Reduction of an array of numbers using the min: method returns the smallest element in the array (alternatively, max: reduction returns the greatest element).
- Reduction of a Boolean array using the | method (*i.e.*, the logical OR) enabling existential quantification. For example, P salary $>$ 3000 \ #| returns true if a pilot has a salary greater than 3000.
- Reduction of a Boolean array using the & method (*i.e.*, logical AND) enabling universal quantification. For example, P salary $>$ 3000 \ #& returns true if all the pilots have a salary greater than 3000, otherwise false.
- Reduction of a Boolean array using the + method which provides the number of objects which check a certain predicate. For example, P salary $>$ 3000 \ #+ returns the number of pilots whose salary is greater than 3000.

Reduction already exists in other object-oriented languages (*e.g.*, inject:into: method in smalltalk, reduce function in Python). However reduction is less used in these languages because of the verbose definition it requires and because object collections in these languages are less frequently present than in OOPAL. In OOPAL the reduction method works in synergy with the other specific high-order methods and idioms promoted by array programming.

Extended Indexing. In array programming, an array can be indexed by a whole array of indices. The translation of this capacity in our OOPAL model involves the array class providing indexing methods able to deal with an array of indices. For instance, if X is {1, 2, 'foo', 'bar', nil, 99, 100}, then X at:{0, 2, 3} returns {1, 'foo', 'bar'}, and X at:{0, 2, 3} put:{-1, -1, -77} modifies X which becomes {-1, 2, -1, -77, nil, 99, 100}. This provides a very general way to designate elements of an array. Severals methods let one conveniently generates arrays of indices for various usages, like selecting a particular region, sorting (see below), shuffling etc.

Sorting. The result of the sort message sent to an array is an array of integers containing the indices that will arrange the receiver of sort in ascending order. For example, if X is {5, 2, 1, 3, 6, 4} then X sort returns {2, 1, 3, 5, 0, 4}. We can then get X in ascending order by indexing it with the result of the execution of the sort method: X at:X sort returns {1, 2, 3, 4, 5, 6}. The advantage of this method is that once we have the ordered index numbers, we can then apply them not only to the original scrambled array, but to any other array of the same size. For example, suppose we want to get an array of pilots sorted by ascending salary. We just have to evaluate the expression P at:P salary sort.

Joins. The OOPAL join operation, loosely named after the join operation in relational algebra, is implemented by the $><$ method of the Array class. For each element, say e, of the receiver, this method computes an array containing the positions of e in the argument. It returns all the arrays packed into an array of arrays. For instance, {1, 2, 'foo'} $><$ {4, 'foo', 1, 'foo', 'foo'} returns {{2}, {}, {1, 3, 4}}. This result means that the first element of the receiver is found at index 2 in the argument, the second element of the receiver is not present at all in the argument and the third element of the receiver is found at indices 1, 3, and 4 in the argument.

[6] Here, we have extended the Smalltalk indexing method for carrying out compression when its argument is a Boolean array.

Suppose we want to obtain, for each pilot in P, the list of all the flights that the pilot is responsible for. That is, we want to construct an array of the same size as P, where each element is an array containing the flights associated with the corresponding pilot in P. The list of flights at index i in the resulting array contains the flights for which the pilot at index i in P is responsible. As shown in Figure 1, the Pilot class does not provide a method (say, "flights") returning the list of flights associated with a pilot. Thus we cannot just write something like P flights. So, how can we navigate from the pilots to the associated flights? It is in this kind of situation that the join method is useful. This method can be used here because the Flight class provides a method (named pilot) that makes it possible to navigate from a flight to its associated pilot. The result can be obtained by combining the join method, an extended indexing, and two message patterns: F at:@ P >< F pilot.

As shown here, the join method makes it possible to easily navigate an object graph without requiring the object model to maintain and provide inverse relationship (*i.e.*, back-links) navigation capacities. Used wisely, this characteristic can simplify the implementation of an object model and allow one to express *ad hoc* queries.

Transposition. The transposedBy: method computes and returns an array which is a transposition of the receiver according to the transposition vector passed as argument. A transposition involves restructuring a multi-dimensional array (represented in OOPAL by nested arrays) so that its coordinates appear in a permuted order. The element of the transposition vector at index i specifies the dimension of the receiver which becomes dimension i in the result. For example, {1, 2, 0} as a transposition vector specifies that the first dimension of the result is the second dimension of the receiver, that the second dimension of the result is the third dimension of the receiver and that the third dimension of the result is the first dimension of the result. Suppose we compute N := (A >< @ (F at:@ P >< F pilot) airplane) @@count. N is an array of arrays that gives the number of flights, for each pilot, for each airplane. That is, to get the number of flights associated with the pilot at index i in P and the airplane at index j in A, we can compute (N at:i) at:j. We can easily transpose N to get an array of arrays (say Nt) which gives the number of flights for each airplane, for each pilot. That is, we can turn rows into columns and columns into rows. To do that we execute Nt := N transposedBy:{1, 0}.

The transposition operation, which enables us to reorganize nested arrays easily, is very useful in OLAP-like data processing.

7.2 A Final Example

Imagine we want to identify the pilots who are in charge of at least one flight for each airplane in the fleet. In a traditional object language like Objective-C, we would write the following code.

Objective-C

```
Airplane *airplane;
Pilot *pilot;
Flight *flight;
NSMutableArray *result = [NSMutableArray array];
NSEnumerator *pilotEnumerator = [P objectEnumerator];
while (pilot = [pilotEnumerator nextObject])
{
   NSEnumerator *airplaneEnumerator = [A objectEnumerator];
   while (airplane = [airplaneEnumerator nextObject])
   {
      NSEnumerator *flightEnumerator = [F objectEnumerator];
      while (flight = [flightEnumerator nextObject])
      {
         if ([flight airplane] == airplane && [flight pilot] == pilot)
            break;
      }
      if (!flight) break;
   }
   if (!airplane) [result addObject:pilot];
}
return result;
```

Below, we present the same query expressed in F-SCRIPT. The following technique is used by the F-SCRIPT code: for each pilot we determine the list of airplanes on which he has at least one flight, using a join. We then use compression to select the pilots for whom the list of airplanes contains as many elements as the total number of airplanes in the fleet.

F-SCRIPT

```
P at:(F at:@ P >< F pilot) airplane @distinct @count = A count
```

8. IMPLEMENTATION NOTES

The OOPAL model has been implemented in F-SCRIPT, an open-source object-oriented scripting language available at www.fscript.org.

8.1 OOPAL in F-SCRIPT

F-SCRIPT is an interpreted and interactive language implemented in Objective-C. F-SCRIPT adopts the Smalltalk syntax. Integration of OOPAL into F-SCRIPT required extending the original Smalltalk syntax and changing the semantics of message send to handle message patterns. F-SCRIPT is based on the Mac OS X native object model. The classes representing arrays and the other basic classes such as numbers or strings are Mac OS X classes. Array programming-specific operations (compression, reduction, etc.) are implemented in the form of methods. All the objects based on the Mac OS X object model can be manipulated using F-SCRIPT. For example, in [38], message patterns are used to manipulate Mac OS X native GUI widgets.

In F-SCRIPT, the main modules related to OOPAL are the parser for the message pattern syntax, the message pattern interpreter, the Array class and the various array programing operations (*e.g.*, reduction, compression). Each module is well isolated from the others, and provides a relatively simple function, making it easy to implement and optimize. For instance, the message pattern interpreter, which could be considered as one of the trickiest modules in the group, has less than two hundred lines of Objective-C code, including the optimizations described in section 8.2.

8.2 Optimizing Performance

The performance and optimization techniques of array programming languages have been widely examined and written about, in particular, due to the frequent use of array programming in the field of high performance financial and scientific calculations. Generally speaking, most of the principles described in the literature can be applied to OOPAL. In this section, we look at a few OOPAL-specific issues linked to the unification of array programming and OOP, and show how we use an optimized message pattern engine and smart arrays in F-SCRIPT. The benchmarks were carried out on a 800MHz PowerPC G4 running Java 1.4.1, GCC 3.1, and F-SCRIPT v1.2.4.

Even though some optimizations are based on optimizing Boolean and number arrays, they are relevant in the specific context of integration of array programming in OOP. Indeed, in the OOPAL model, operations on number and Boolean arrays are very frequent during the manipulation of *any object*. The fundamental operations offered by OOPAL for manipulating any type of object (compression, joins, reduction, transposition ...) and the various associated

Table 1: Memory footprint of an array containing five million numbers

F-SCRIPT without smart arrays	176 MB
F-SCRIPT with smart arrays	40 MB
C	40 MB

idioms are intrinsically based on the generation and manipulation of number and Boolean arrays. For example, the F-SCRIPT expression P age $<$ 35 & (P salary $>$ 3000) \ #| which determines whether any pilot under the age of thirty-five has a salary greater than 3000, generates and uses five number and Boolean arrays.

Optimized Message Pattern Engine. OOPAL message patterns may be compiled or interpreted. If you have a code generator and a compiler (whether static or just-in-time), you can translate them in compiled native code implementing nested loops. If you do not have these modules or if you want to avoid them as they are both cumbersome and complex, you can interpret the message patterns at run-time using a message pattern interpreter. To avoid adversely affecting performance, optimized (i.e. compiled) implementations of the most common patterns can be integrated in the interpreter itself. In F-SCRIPT, the message pattern interpreter contains optimized implementations for several patterns, including @, @:, @:@ and @1:@2. Benchmarks show that, in F-SCRIPT, sending a simple message to each element of an array using the @ pattern is 23 percent faster with this optimization.

Smart Arrays. OOPAL offers the language user the notion of heterogeneous object arrays. But in practice, in most common cases of use, most arrays are in fact homogenous. In addition, as we already mentioned, array programming-specific operations produce a majority of number and Boolean arrays. These properties enable the implementation of the main OOPAL model optimizations. At the base of these optimizations are what we refer to here as *smart arrays*: arrays capable of *dynamically* adapting at run-time their internal representation to their content so as to optimize the operations carried out on the arrays and the memory footprint. In F-SCRIPT the class Array acts as a facade that hides the fact that different optimized classes are used to specifically handle certain arrays such as Boolean and numerical arrays. In an array language like F-SCRIPT, this optimization is very useful because of the pervasiveness of homogenous arrays. In F-SCRIPT number and Boolean arrays use optimized internal representations and operations:

- Number and Boolean arrays use an internal representation corresponding to native data arrays of the platform. For example, in conventional 32-bit architectures, a double-precision number array uses, as internal representation, a contiguous memory space made up of 64-bit words where each word represents a double-precision number in native format. Table 1 shows the memory footprint, in mega bytes, of an array which contains five million double-precision numbers.

- Sending of messages to elements of these arrays is automatically bypassed and replaced by the direct application of native operations. For example, if X is a number array and y a number, the expression X + y does not generate costly sending of messages but, rather, the application of a native addition operation for each element in X.

- Several operations are optimized to take advantage of smart arrays. For example, reduction of a Boolean array by a logical OR will stop as soon as a *true* value is found in the array.

- Dynamically adapting arrays and the use of optimized operations and representations are transparent for the user who only manipulates object arrays, except when talking about the object identity of numbers, which can be changed by optimized representations. When we place an object representing a number in a smart array and then retrieve it, the result is an object with the same value as the object we placed in the array. However, there is no guarantee that it will be the same object in memory. This modification of the conventional semantics of object arrays has few consequences, given that the notion of object identity is almost never used with numbers.

- The way in which arrays dynamically adapt their internal representation is based on heuristics that make it very fast. For example, arrays that have been heterogeneous once in their lifetime never check their content to determine if they could benefit from a homogeneous-optimized internal representation. This heuristic is based on a hypothesis that we can practically summarize as follows: heterogeneous arrays rarely become homogeneous. More generally, the internal representation that an array possesses at a certain point in its lifetime tells something about the way the array has been used so far. This "something" helps the array to determine the likely and unlikely evolutions of its content and usage. This enables it to implement the dynamic adaptation mechanism efficiently.

- The F-SCRIPT smart array system is designed to be extensible. It is based on an object-oriented framework which facilitates the definition of new types of internal representations for arrays and the implementation of operations optimized for these representations. It is thus possible to define optimized operations and representations for arrays containing objects other than numbers and Booleans.

Combined with smart arrays, the OOPAL model ensures both code conciseness and good performance for array expressions. It helps F-SCRIPT to overcome the handicaps of being interpreted, dynamically typed, and a "pure object" language in which any number or Boolean is manipulated by sending a message. For instance, table 2 compares the code and time in seconds needed to add two number arrays element-wise in F-SCRIPT, Java and C. Each array contains five million of double precision numbers.

Optimizing for Hardware. Array programming has been noted to be well suited to various hardware architectures such as vector processors and parallel architectures (see [1, 5, 9, 12, 14, 15, 24]). For instance, short vector architectures, which are found in the form of SIMD (Single Instruction, Multiple Data) units in mainstream microprocessors, are well suited to the ultra-fast processing of entire arrays. Array programming, which is intrinsically SIMD, makes it possible to use vector architectures in order to to optimize the OOPAL model. Even in a pure object-oriented language, smart arrays allow processes relating to arrays to be implemented in an optimized manner - transparent to the user - by using native SIMD instructions. For example, let us look at the expression P age $<$ 35 & (P salary $>$ 3000) \ #|, which determines whether any pilot under the age of thirty-five has a salary greater than 3000. Of the six operations included in this expression (five message patterns and a reduction), four could be directly implemented using the vector unit: this involves two comparisons, a logical AND and a reduction. On an Altivec unit, the instructions vec_cmpgt, vec_cmplt, vec_and and vec_any_eq could be used. However, it should be noted that the optimization of basic operations on arrays does not

Table 2: Code and time required to add two arrays containing five million numbers each

	Code	Time

Java using objects	final int size = 5000000;
	ArrayList C = new ArrayList(size);
	for (int i = 0; i < size; i++)
	C.add(new Double(((Double)(A.get(i))).doubleValue()
	+ ((Double)(B.get(i))).doubleValue()));
		7 s

Java using primitives	final int size = 5000000;
	double[] C = new double[size];
	for (int i = 0; i < size; i++) C[i] = A[i]+B[i];
		0.5 s

C	const int size = 5000000;
	double *C = (double *)malloc(size*sizeof(double));
	for (i = 0; i < size; i++) C[i] = A[i]+B[i];
		0.5 s

F-SCRIPT	C := A + B	0.5 s

guarantee that performances will be improved. Indeed, without some kind of code consolidation (see section 8.3 *Loop Fusion and Array Reduction*), each of these operations requires a full iteration on one or more arrays and, therefore, exchanges of data between the processor and the memory. If memory access is slow in relation to processor speed, then a bottleneck will develop, which detracts from the full processing power of the vector unit [43]. For example, F-SCRIPT uses the vector unit to implement the basic logical operations on arrays of Booleans. On machines with slow memory bus, the performance improvements are fairly slight. In practice, it is the most complex operations (those which require many primitive instructions with each iteration on each element of the arrays being processed), which reap the greatest benefits from the use of the vector unit [40]. For the others (like basic logical operations), the performance improvement is closely related to the memory access speed, which can vary greatly from architecture to architecture.

8.3 Other Possible Optimizations: Loop Fusion and Array Reduction

Loop fusion and array reduction minimize the number of loops and the creation of temporary arrays during expression evaluations.

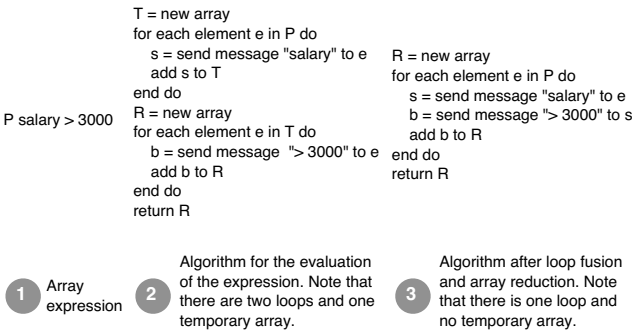

Figure 7: Example of loop fusion and array reduction in an array/object-oriented context.

Note that loop fusion and array reduction have not yet been implemented in F-SCRIPT. These are very important techniques used by some array languages to improve performance [31]. However, these optimizations cannot be applied blindly. They change the order of operation and this can modify the semantics of the evaluated expressions. In some cases, however, the language has enough information to determine that loop fusion retains correct evaluation semantics. This is the case when the language can determine that sub-expression evaluations do not interact. However, the characteristics of object-oriented programming (polymorphism, extensibility, etc.) make this difficult.

Within the scope of an interpreted language with no static typing, in some situations, smart arrays can enable the interpreter to effectively determine the nature of the operations (in particular, determining that the operations do not modify the state of objects) thus giving the interpreter the possibility of implementing loop fusion and array reduction. In statically typed object-oriented languages, loop fusion and array reduction can sometimes be applied at compile time, as shown by the Expression Template system [49, 51], which uses the C++ template engine to perform loop transformations at compile-time.

Some object models facilitate the implementation of loop fusion and array reduction: [16] describes a model which enables reasoning about the disjointness of computational effects within the scope of object-oriented programming and provides an extensive bibliography on the subject.

9. REAL WORLD APPLICATIONS

The OOPAL model is used, through F-SCRIPT, in various fields such as data analysis, game development, application scripting, or software debugging. In this section we give two examples of real-world usage that are independent of our research.

OOPAL in Astrophysics. F-SCRIPT is used at the department of Astronomy & Astrophysics of the University of Toronto to interactively filter and manipulate data coming from the Hubble Space Telescope. Central to the objet model of this application is a Galaxy class which offers many methods returning galaxy's properties. Professor Roberto Abraham reports:

> F-SCRIPT lets one easily do pretty complex data mining to drill down through samples of thousands of galaxies distributed throughout a very large parameter space in order to isolate only those galaxies of interest. Amongst the infinite number of things we might want to do is restrict some analysis to all galaxies with a certain class with a median area above some value, and then work out the mean value of a bunch of galaxy properties. One certainly can't anticipate beforehand what one will want to do with the data. Since manipulation of the data needs to be general, and the data analysis is inherently array based, embedding F-SCRIPT is perfect for this application. It replaces having to filter million of little ASCII text files with AWK and Perl, and

since the F-SCRIPT syntax automatically threads over arrays no serious programming is needed to do really complicated stuff.

For instance, here is an F-SCRIPT expression that returns the isophotal magnitudes of the galaxies that have a major axis lesser than 50: (galaxies at:galaxies majorAxis $<$ 50) isophotalMagnitude. This example shows that the designer of the galaxy model expresses the queries within the metaphor he created and is not forced to manipulate naked data.

OOPAL in Music Theory. F-SCRIPT is used at the Technical University of Berlin by the Interdisciplinary Research Group for Mathematical Music Theory, in the context of the Rubato project and the OpenMusic/Humdrum/Rubato (OHR) project. F-SCRIPT is used to aid workflow between Music analysis tools making it possible to specify music-theory material (for instance, a harmonic analysis theory) and to exchange it between existing tools (*i.e.*, OpenMusic and Rubato). OpenMusic is able to automatically generate F-SCRIPT code which manipulates Rubato's objects. Among other things, F-SCRIPT is used to implement some music-related algorithms. For instance, an F-SCRIPT program has been developed that makes it possible to test a new approach to harmonic analysis suggested by Fred Lerdahl within the existing application framework of a Riemannian functional analysis implementation (the Harmo-Rubette). Jörg Garbers, member of the Music Theory Research group, notes that message patterns are extensively used as well as reduction and iota[7] operations. [17] contains further description of how F-SCRIPT is used in this context.

10. RELATED WORK

Several teams have explored integration of object technology and array programming. In most cases, their goal has been to study how object technology could enhance an existing array language. See [2, 4, 7, 8, 11, 18, 19, 33, 41, 44, 48]. OOPAL takes a rather different approach as it involves bringing the notions of array programming to the object world. Its goal is therefore to determine the minimal set of modifications that must be made to the traditional object model in order to take advantage of the possibilities of array programming. To our knowledge, little research has been carried out in this area.

Marcel Weiher [52], proposes a powerful model based on dynamic wrappers (a.k.a. trampolines), which enables high-order messaging. This provides an alternative approach to OOPAL's message pattern for high-order messaging. Unlike OOPAL, this model can be attached to a dynamic object language without needing to modify its syntax or semantics. However, it does not provide a notation as convenient as the one offered by OOPAL.

Several libraries provide array programming operations for existing object-oriented languages. For instance, SmartArrays [10], developed by APL gurus, provides an advanced array programming library for C++, Java, and C#. Other examples include the Numerical Python package, and Numarray [21], its successor, which add array programming operations to Python, and the Blitz++ library [50] for C++. The fundamental difference between the OOPAL model and such libraries is that these libraries are oriented towards numeric computing, whereas OOPAL attempts more fundamental integration between object-oriented programming and array programming to support the manipulation of any kind of object.

[7] *iota* is an indices generator, *i.e.*, a method that allows one to easily create arrays of integers. It is widely used in array programming, and has been adapted to OOPAL.

The confrontation between object-oriented databases and relational databases has led to the development of object query languages, sometimes based on sophisticated object algebras and calculi, which provide high-level object-oriented querying models. However, these developments, at least for the moment, have not influenced mainstream object-oriented programming languages such as C++, Java, C#, or Smalltalk. On the contrary, in their most recent incarnations (*e.g.*, JDO Query Language, EJB Query Language), object query languages adopt quite a low profile. They are merely used as a minimalist interface to an underlying database, for bootstrapping the data navigation and manipulation process (*i.e.*, getting some elements of the object graph out of the database) which is then carried out with the host object-oriented programming language. While object query languages are mainly designed around the interaction with persistent objects stored in a database, the OOPAL model is primarily designed to interact with instantiated objects lying in memory. Indeed, the problem we are tackling with OOPAL is to provide a higher-level programming model in the context of OOP, not a database query language. One important consequence of this difference in focus is the support for encapsulation. For performance reasons, most of the query languages provided by object-oriented databases or object/relational mapping tools break encapsulation. For example, the current version of JDOQL [46] does not support method invocation of persisted objects but only offers direct access to instance variables. The performance problem that justifies breaking encapsulation is due to the fact that database query languages do not actually manipulate objects but object representations stored on disk, which is a different thing. In such a case, the use of indexes and the minimization of instantiations, which can be achieved by breaking encapsulation, are required in order to achieve good performance. On the contrary, in the context of a general purpose object-oriented language, encapsulation is of paramount importance, and is well supported by OOPAL.

High-level object-oriented languages like Smalltalk or Python commonly provide sophisticated collections classes associated with high-level operations and constructs. A key difference between this approach and OOPAL is that the latter allows one to think in terms of whole sets of objects, while conventional object collection protocols require thinking in terms of iteration over collection elements. For instance, compare the code needed to generate an array containing the names of all the pilots:

- Smalltalk: P collect:[:aPilot| aPilot name].
- Python: [aPilot.name() for aPilot in P].
- Ruby: P.collect {|aPilot| aPilot.name}
- F-SCRIPT: P name.

OOPAL's association of message patterns and array programming operations subsumes, for most purposes, these common high-level models by making it possible to express object manipulations in a more readable way and with less code. For instance, the collection-protocol-based Smalltalk code to generate an array that contains the names of all the pilots whose age is greater than 50 is:

(P select:[:aPilot| aPilot age $>$ 50]) collect:[:aPilot| aPilot name]

The same is expressed using the OOPAL model with:

P name at: P age $>$ 50

As well as showing radical gains in readability and conciseness, this example illustrates the fact that OOPAL helps to decrease the use of lambda expressions. This property of array programming was noted by John Backus in [3]:

> We owe a great debt to Kenneth Iverson for showing us that there are programs that are neither word-at-atime nor dependent on lambda expressions.

We should also note that OOPAL's array expressions allow combining multiple arrays while traditional internal iterators like select: only iterate over a single collection[8].

Finally, it should be noted that several array extensions to existing programming languages have been produced, sometimes with broad acceptance, as with Fortran which has incorporated array programming features in recent versions of the language and in the official standard (see [35]). In some cases, similarities with OOPAL can be observed. For instance the @ sign in a message pattern is reminiscent of the α sign applied to a function and of the $\cdot$ sign in the context of an α-factored expression in Connection Machine Lisp [22].

11. CONCLUSION

On one hand, object-oriented programming provides good support for data *modeling* but it falls short for the expression of manipulations of *entire sets* of objects. This lack of expressiveness of the object-oriented approach was one of the reasons why some relational advocates saw object-oriented databases as a twenty-year step backwards. On the other hand, array programming supports the manipulation of entire sets of data and avoids the use of explicit loops, but does not support objects.

This article presented the OOPAL model that defines the integration of array programming in object-oriented programming. It is based on an extension of the concept of method invocation which implies a slight syntactic extension, and the addition of certain array manipulation methods, which do not call for the modification of the target language's syntax.

The manner in which F-SCRIPT handles encapsulation, inheritance, and polymorphism has not been addressed in this paper due to the similarity with other dynamic language such as Smalltalk. In the OOPAL model these concepts are orthogonal to array programming and don't need to be specifically re-designed to fit into the world of array programming. This point is clearly shown by the fact that F-SCRIPT allows standard Objective-C objects to be manipulated.

In OOPAL, array programming and OOP fit well together. Not only can these two programming models benefit from each other's advantages, but they act synergically. While array programming makes it possible to write code with few explicit loops, OOP's dynamic binding decreases the use of explicit conditional control structures. It just so happens that the need to use explicit conditional control structures usually hinders the use of array programming, because it makes it harder to express array-oriented algorithms. OOP, which favors a coding style that is free from these structures, naturally allows for extended use of array programming.

The array programming world is rich. It is full of intriguing and powerful data manipulation techniques and idioms, developed and refined by thousands of creative people over nearly half a century of intensive research and real-world system design. By integrating array programming in object-oriented programming, the OOPAL model shows how object-oriented programming can benefit from a large proportion of these developments. This provides a high-level notation which makes it possible to easily express complex object manipulations. The real-world applications developed with F-SCRIPT have shown that the integration of array technologies with object-oriented programming adds a considerable amount of power and ease-of-use to the latter.

[8] In [30], Thomas Kühne proposes a new iteration pattern, named "transfold", which does not suffer from this lack of flexibility.

12. ACKNOWLEDGEMENTS

We would like to thanks Noury Bouraqadi, Jörg Garbers, Ralph Johnson, Joseph R. Kiniry, Brid Marire, Oscar Nierstrasz, Hannah Riley, and Roel Wuyts for the their feedback on early versions of this article.

13. REFERENCES

[1] E. Albert, K. Knobe, J. D. Lukas, and J. Guy L. Steele. Compiling Fortran 8x array features for the connection machine computer system. In *Proceedings of the ACM/SIGPLAN conference on Parallel programming: experience with applications, languages and systems*, pages 42–56. ACM Press, 1988.

[2] M. Alfonseca. User interfaces with object-oriented programming in APL2. In *Proceedings of the conference on Designing the future*, pages 1–11. ACM Press, 1996.

[3] J. Backus. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[4] J. P. Benkard. An epistemology of APL. In *Proceedings of the APL98 conference on Array processing language*, pages 76–90. ACM Press, 1998.

[5] R. Bernecky. The role of APL and J in high-performance computation. In *Proceedings of the international conference on APL*, pages 17–32. ACM Press, 1993.

[6] P. Berry. *APL 360 Primer*. IBM, third edition, 1971.

[7] B. Best. Object-oriented programming and APL computer-language. http://www.benbest.com/computer/oopapl.html.

[8] P. Bottoni, M. Mariotto, and P. Mussio. LiSEB: a language for modeling living systems with APL2. In *Proceedings of the international conference on APL : the language and its applications*, pages 7–16. ACM Press, 1994.

[9] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka. Fortran 90D/HPF compiler for distributed memory MIMD computers: design, implementation, and performance results. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 351–360. ACM Press, 1993.

[10] J. A. Brown and J. G. Wheeler. SmartArrays for the APL programmer. In *Proceedings of the 2002 conference on APL*, pages 50–60. ACM Press, 2002.

[11] R. G. Brown. Object oriented APL: an introduction and overview. In *Proceedings of the international conference on APL-Berlin-2000 conference*, pages 47–54. ACM Press, 2000.

[12] T. A. Budd and R. K. Pandey. Compiling APL for parallel and vector execution. In *Proceedings of the international conference on APL '91*, pages 80–87. ACM Press, 1991.

[13] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder, and W. D. Weathersby. ZPL: A machine independent programming language for parallel computers. *IEEE Transactions on Software Engineering*, 26(3):197–211, 2000.

[14] W.-M. Ching. Automatic parallelization of APL-style programs. In *Conference proceedings on APL 90: for the future*, pages 76–80. ACM Press, 1990.

[15] W. M. Ching and A. Katz. An experimental APL compiler for a distributed memory parallel machine. In *Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, pages 59–68. ACM Press, 1994.

[16] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the 17th ACM conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 292–310. ACM Press, 2002.

[17] J. Garbers. Kit-mamuth: Zwischenbericht softwareentwicklung. Technical report, Technical University of Berlin, 2001.

[18] M. Gfeller. Object oriented programming in AIDA APL. In *Conference proceedings on APL as a tool of thought*, pages 164–168. ACM Press, 1989.

[19] J. J. Girardot and S. Sako. An object oriented extension to APL. In *Proceedings of the international conference on APL*, pages 128–137. ACM Press, 1987.

[20] K. Glazebrook and F. Economou. PDL: The Perl Data Language. *Dr. Dobb's Special Report*, 1997.

[21] Greenfield et al. Numarray, an open source project, 2002. http://stsdas.stsci.edu/numarray/numarray.pdf.

[22] J. Guy L. Steele and W. D. Hillis. Connection machine lisp: fine-grained parallel symbolic processing. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 279–297. ACM Press, 1986.

[23] H. Hellerman. Experimental Personalized Array Translator System. *Communications of the ACM*, 7(7):433–438, 1964.

[24] W. D. Hillis and J. Guy L. Steele. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[25] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: the story of Squeak, a practical Smalltalk written in itself. In *Proceedings of the 1997 ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages and Applications*, pages 318–326. ACM Press, 1997.

[26] K. E. Iverson. Computers and mathematical notation. http://www.jsoftware.com/jbooks_frame.htm.

[27] K. E. Iverson. Math for the layman. http://www.jsoftware.com/jbooks_frame.htm.

[28] K. E. Iverson. *A Programming Language*. John Wiley and Sons, 1962.

[29] K. E. Iverson. Operators. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):161–176, 1979.

[30] T. Kühne. Internal iteration externalized. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 329–350. Springer-Verlag, 1999.

[31] E. C. Lewis, C. Lin, and L. Snyder. The implementation and evaluation of fusion and contraction in array languages. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming Language Design and Implementation*, pages 50–59. ACM Press, 1998.

[32] K. J. Lieberherr and A. J. Riel. Contributions to teaching object-oriented design and programming. In *Conference proceedings on Object-Oriented Programming Systems, Languages and Applications*, pages 11–22. ACM Press, 1989.

[33] R. MacDonald. Bob brown and object oriented APL. http://www.torontoapl.org/ga/ga9605/bbrown.txt.

[34] C. D. McCrosky, J. J. Glasgow, and M. A. Jenkins. Nial: A candidate language for fifth generation computer systems. In *Proceedings of the 1984 annual conference of the ACM on The fifth generation challenge*, pages 157–166. ACM Press, 1984.

[35] M. Metcalf and J. K. Reid. *Fortran 90/95 explained (2nd ed.)*. Oxford University Press, Inc., 1999.

[36] P. Mougin. *F-Script Guide*, 1999. http://www.fscript.org.

[37] P. Mougin. High-level object oriented programming with array technology. In *Proceedings of the international conference on APL-Berlin-2000 conference*, pages 163–175. ACM Press, 2000.

[38] P. Mougin. Scripting cocoa with F-Script, Nov. 2001. O'Reilly Network, http://www.oreillynet.com/pub/a/mac /2001/11/30/scripting_fscript.html.

[39] NialSystem Corp. About the Nial language. http://www.nial.com/AboutNial/AboutNial.html.

[40] I. Ollmann. Altivec tutorial v1.2, 2002. http://www.simdtech.org.

[41] G. Reichard. Is there a way of combining array-processing and object-oriented programming? In *Proceedings of the 2001 conference on APL*, pages 83–86. ACM Press, 2001.

[42] h. Research Systems, Inc. IDL, the Interactive Data Language.

[43] J. Sébot and N. Drach-Temam. Memory bandwidth: The true bottleneck of SIMD multimedia performance on a superscalar processor. In *Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, pages 439–447. Springer-Verlag, 2001.

[44] D. A. Selby. An object-oriented APL2. In *Proceedings of the international conference on APL : the language and its applications*, pages 179–184. ACM Press, 1994.

[45] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Conference proceedings on Object-Oriented Programming Systems, Languages and Applications*, pages 38–45. ACM Press, 1986.

[46] Sun-Microsystems. Java Data Object specification,1.0, 2002.

[47] K. Systems. K user manual, 1998. http://www.kx.com/download/documentation.htm.

[48] N. J. Thomson. *J - The Natural Language for Analytic Computing*. Research Studies Press, 2001.

[49] T. L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[50] T. L. Veldhuizen. Arrays in blitz++. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

[51] T. L. Veldhuizen and M. E. Jernigan. Will C++ be faster than Fortran? In *Proceedings of the 1st International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'97)*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

[52] M. Weiher. Higher Order Messaging (HOM). http://www.metaobject.com/papers/HOM-Presentation.pdf.

[53] R. G. Willhoft. Comparison of the functional power of APL2 and Fortran 90. In *Proceedings of the international conference on APL '91*, pages 343–357. ACM Press, 1991.