Distributed Prograph

B. Lanaspre and H. Glaser

Introduction

Prograph [2] is a visual programming language which uses dataflow as its model; it has an object-oriented class system which provides modularity and code reusability, and the whole is delivered in an integrated environment. The aim of the Distributed Prograph project is to enhance this environment to deliver distributed computing.

Graphical dataflow gives programmers a clear view of the potential for exploitation of concurrency and so the Prograph language appears to give some leverage for the programming of parallel or distributed systems. A small number of annotations to the language should be introduced to indicate parallelism but the programmer's model should change as little as possible in the presence of distributed execution. The pure dataflow model is free from side-effects and global variables. Prograph implements a modified version of dataflow principles in which freedom of side-effects and absence of global variables are no longer guaranteed.

Prograph features

Visual Dataflow

Prograph is a visual dataflow language. A computation or *method* consists of a sequence of one or more *cases* (see Figure 1). A *case* is a dataflow diagram made up of boxes called *operations* and connected by *links*. The top boundary of the diagram is established by an *input* operation and the bottom one by an *output* operation with, in between, a graph of operations possibly connected by data and synchronisation links (*synchros*).

Conceptually the computation is driven by the availability of data. When all the inputs of an operation (or *terminals* in Prograph terminology) are available, the operation can be executed. The execution produces an *execution message* (*fail* or *succeed*) and the results are output on the *roots* of the operation box. The flow of execution may also be effected by *controls* attached to operations.

When a method is called, execution commences in the first case, it proceeds until a control is activated or it succeeds completely. Activation of a control may result in the execution being stopped, the method being failed or the flow of control being transferred to the next case where the computation resumes at the input operation.

An operation on the graph can be:

- a call to some compiled code (*primitive*).

- a call to a Prograph method. A method can be a *Local*, *Universal* or *Class-based* method. A Local is a method within a method. The visibility of the Local is limited to the scope of the containing method. A Universal Method is a method which is not associated with any class.

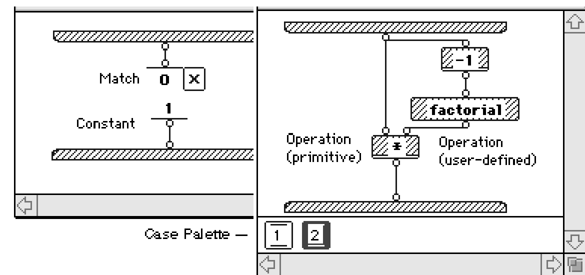

Figure 1: A User-defined method

The behaviour of an operation can be modified by a *multiplex* annotation. A *repeat* annotated operation executes repeatedly until a control is activated. Terminals and roots may also be annotated. A *list* annotated terminal indicates that the argument on this terminal will be a list and that the operation should be applied to every element of the list. A *loop* annotation creates a cycle whereby the value coming out of a root is fed back into a terminal until the execution finishes. A *partition* operation converts a predicate into a filter operation which splits an input list into two lists.

The *input* and *output* operations have a special status. Both operations are nameless. The input operation has, by definition, no terminals and the output operation no roots.

The pure dataflow model prohibits alteration on data objects. If a data object is to be modified, a copy of it is made instead. However, application of the strict dataflow principles would result in performance costs when complex data structures are involved. For the sake of efficiency, Prograph implements a modified version of dataflow principles. *Primitive type* data objects are copied but instances of classes are modified "in place".

It is claimed that the visual paradigm and the data-driven semantics of the language make the logic and the evaluation more understandable and help the programmer to grasp the potential for parallelism. The system described in [1] is also based on visual programming and data-driven computation.

Object-orientation

Wegner [4] distinguishes three generic features for object-oriented systems: *objects*, *class* and *inheritance*. An object encapsulates both state and the operations to manipulate that state. Objects are instances of classes. A class is an abstract template specifying the attributes and the operations associated with all its instances. With inheritance, a *subclass* inherits the behaviour of its *parent* class; the parent class may, in turn, have inherited some of its behaviour from a parent class.

Unlike Smalltalk [3], Prograph has not adopted "the everything is object" philosophy. With Prograph, the data flowing across the arcs of the dataflow graph can be of a Prograph primitive type: *boolean*, *external*, *integer*, *list*, *none*, *null*, *real*, *string* or *undefined*. Alternatively, they can be instances of user-defined classes. Thus a distinction is made between a static set of Prograph primitive types and an extensible class hierarchy. Prograph is a

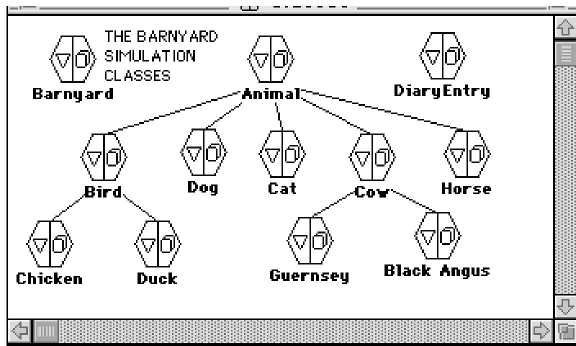

Figure 2: Class icons and inheritance trees

dynamically typed language. In dynamically typed languages, the type information is not associated with variables but with values. This means that the same variable can store successive values of different types. Dynamic typing gives Prograph a greater flexibility.

Attributes defined by a class can be divided into two categories: the *class variables* of any class are shared by all its instances, *instance variables* are private to each instance. Prograph provides *persistents*. Persistents are named elements which can hold any value. Class variables and persistents are similar to global variables in procedural languages. It must be noted that Prograph makes a limited use of variables: they are reserved for instance and class variables as well as persistents.

The Prograph class system supports *single inheritance*; that is each class has, at most, one parent class. The parent class may also inherit from a single superclass. The inheritance chain can be of arbitrary length. The at-most is significant in the sense that the Prograph class hierarchy is “a forest of trees” and not “a single tree” as in Smalltalk in which any pair of classes have at least the *Object* class as common ancestor. It is important to note that if the both the class and the instance attributes are inherited, the default values for these attributes are not. The methods associated with the superclass are inherited.

Prograph methods

The Universal methods share a single name space and each class can be seen as a distinct name space on its own. Subclasses inherit the behaviour of their superclasses. If the behaviour needs to be altered, a subclass may override the incriminated methods. The binding of a named behaviour to a method is called *method dispatching*. A method is said to be applicable to a class if the class defines or inherits the method from a superclass. Failure to find the named method is sanctioned by a runtime error. There are four ways of referencing a method:

1. A *universal* reference. The operation is associated with a universal method.

2. A *data-determined* reference. The behaviour invoked is the method applicable to the class of the instance flowing into the left-most terminal of the operation. If there is no applicable method, a universal method or a primitive may be called.

3. An *explicit* reference. The operation name consists of both a class name and a method name. The method called is the method applicable to the named class.

4. A *context-determined* reference. The method is the method applicable to the class of the method containing the operation. Therefore, it is impossible to name operations with a context-based reference in the context of an Universal method. A context-determined reference can be *super* annotated. The method called must be applicable to the superclass of the class to the method containing the operation.

A *Simple class* method is a method of arbitrary arity associated to a class. A class also defines two sets of attributes, the class and the instance attributes. Classes, unlike instances, can be referenced by name. The default *Get* operation returns the value of an attribute; the name of the operation is the name of the attribute to be accessed. The arity of the Get is fixed: one input and two outputs; the value of the leftmost output is that of the input, the second output returns the value of the attribute. Similarly, the value of an attribute can be changed using the default *Set* operation; the arity of Set is two inputs and one output; the name of a class or an instance flows into the leftmost input, the value of the attribute in the second input, the output returns the value fed into the first input.

It might be, in some cases, necessary to extend the behaviour of the default Set or Get methods by defining a new method. The new method must have the same arity as the default operation. A data- determined reference must be used in order to call a user-defined Get or Set method instead of the default one. A Set or a Get method may have a name that does not correspond to any attribute in the class. A *virtual* attribute is thus defined.

A default *Initialisation* method also comes with each class. It can be overridden by the user. The user-defined method arity is required to match that of the default method.

A model for Distributed Prograph

The dataflow metaphor

In the dataflow model the execution is driven by the availability of data and the sequencing is only constrained by data dependencies. This model does not require an explicit management of parallelism, it is implicit. Operations on the dataflow graph are units of potentially fine-grain parallelism.

In a hardware dataflow machine, parallelism is achieved by having several functional units executing operations simultaneously. The metaphor of a virtual dataflow machine can be used to explain the Distributed Prograph model. Parallelism is implicit in the dataflow model. However for safety and performance reasons it is felt that the programmer should retain control of the parallelism in the Distributed Prograph model. The safety concern arises from the existence of side-effects and global vari-

ables. On the performance side, operation-level parallelism is too fine-grain for Distributed Prograph.

Distributed Prograph aims at executing a single application on a set of machines. The application is initiated on the user machine. Annotated portions of the application graph can then be sent to remote processors for execution. The machine acts as the sequencing and update unit. It is responsible for distributing tasks in the first place and all results are ultimately returned to it. It is however possible for remote processors to send tasks of finer granularity to other remote processors. The dataflow graph is thus dynamically expanded into a tree of tasks with the user application at the root of the tree.

Distribution issues

The pure dataflow model is side-effect free. However, Prograph has both side-effects and global variables. In the current version of Prograph, the operations on the dataflow graph are executed according to a serial schedule of execution; that is, each operation constitutes an atomic unit of execution. This property means that the execution is correct in most cases; the programmer has, sometimes, to impose a special ordering for the execution of the operation and this is typically the case when of the operation is a test or an I/O operation. The concurrent execution of operations might lead to interferences. Distributed execution would introduce further synchronisation requirements.

Through the existence of global variables, some values are accessible in the scope of a method without being passed as arguments to the method. Operations may induce side-effects on their arguments and on global state. When a computation is distributed over several address spaces, it is necessary to ensure that the global state is kept consistent and that the side-effects are implemented properly.

Distribution mechanisms

The Distributed Prograph model of execution should remain as close as possible to the original model. Distribution encompasses four broad categories of activities:

- Preparation of an operation packet
- Operation scheduling
- Execution of an operation by a remote processor
- Return of the results

These four activities are discussed in the following subsections

Operation packet

The operation packet is the information sent for the execution of an operation by a remote processor. It comprises the name of the operation along with the arguments of the operation. The method called by the operation must be available in the address space of the packet recipient. For class-based methods, the class to which the method belongs and all the inheritance chain must be available. For context-determined references, it is important to know the class of the method which contains the operation.

An operation may access global variables during its execution. Such accesses must be anticipated at compile time. The values of the global variables that may be needed are sent with the operation packet. Objects can be structures of arbitrary complexity. The object graph must be searched recursively. Cyclical structures must be dealt with.

Operation scheduling

Once the operation packet has been readied, a target processor must be selected. The solution to this allocation problem will to a large extent determine the efficiency of Distributed Prograph. Operations selected for remote execution can be nested. A processor which has received an operation packet may in turn send an operation to another processor.

Remote execution of an operation

Operation packets are received by the remote processors and pooled before their execution. The remote processor may be required to update some global variables in its address space. This should be carried out before executing the operation. Execution of an operation on a remote processor might modify the value of some global variables and of the arguments of the operation. These modifications must be anticipated statically so that the modified values be sent along with the results of the operation. The execution message (e.g. *fail* or *succeed*) of the operation must also be returned.

Failures are more difficult to detect and to recover in a distributed environment than in a centralised one. Ideally, the processor should be able to recover from a failure and be able to report it to the sender of the packet. This would probably require some substantial modifications to the Prograph language. The dynamic type system of Prograph makes it more difficult to detect type errors at compile time. Some failures can be the consequences of distribution: method or class not available in the program of the worker. Bad programming logic has potentially more damaging effects than in a single machine environment. A badly designed and tested method can result in an infinite loop or stack overflow. The processor then appears to have failed. Network communication can also fail.

Return of the results

From the programmer's point of view, Distributed Prograph must retain the data driven semantics of Prograph. The current version of Prograph supports a single thread of execution. This notion of single threaded execution is easily understood when Prograph code is executed in interpreted, trace mode. At any time, a single operation is active on the dataflow graph. An operation can become active only if the previous one has completed, even when there are no dependencies or synchronisation constraints between the two operations. With its distributed version, Prograph will evolve to a multi-threaded model of ex-

ecution. Methods selected for parallel execution can be activated simultaneously. To retain a data-driven semantics, the completion of an operation blessed for parallel execution will be considered effective only if the results of the operation are available, or alternatively, the operation is declared failed. If the results are not yet available then the operation should block. To maximise the use of the processor, the blocking of an operation should trigger a thread switch so that the processing resource might be used for some other tasks. These tasks might include processing an operation that has been pooled before being exported, retrieving the results for another exported method.

Benefits of the model

It is felt that the data-driven model of program sequencing chosen for Prograph will help to keep the replication of global variables and the implementation of side-effects manageable. The copy of a global variable in a remote processor's address space need be updated only if an operation accessing this variable is sent to the remote processor. The synchronisation of the update operations is done by the exchange of messages between processors.

The model does not tackle the problem of interfering concurrent operations. The difficulty of ensuring the consistent scheduling of operations should not exceed that of the current version of Prograph. The benefit of graphical dataflow is that the programmer can more easily see when parallel execution is inconsistent.

The model should relieve the programmer of the task of mapping parallel activities to processors. It should also prove scalable, the same program should be able to execute on a single processor as well as on a collection of autonomous processors, albeit with a performance difference.

Conclusion

In this paper, the motives behind Distributed Prograph have been explained and the targets of the projects set. The features of the language have been reviewed in depth and a programming model for Distributed Prograph proposed. In order to lay down a clear road map for the project, the issues involved in the implementation of Distributed Prograph have been categorised and covered with some detail. Work in progress has reached a stage where a prototype static analyser should be available soon. The further work on the main implementation is now funded by the ESPRC under the GraphICSLA project.

References

1 Alvisi, L., Amoroso, O., Babaoglu, O. Baronio, A., Davoli, R. and Giachini, L.A. (1992). "Parallel Scientific Computing in Distributed Systems: The Paralex Approach". Technical Report UBLCS-92-2, Laboratory for Computer Science, University of Bologna.

2 Cox, P.T., Giles, F.R. and Pietrzykowski, T. (1989). "Prograph: a step towards liberating programming from textual conditioning". 1989 IEEE Workshop on Visual languages, pp. 150-6.

3 Lalonde W.R. and Pugh J.R. (1990). Inside Smalltalk. Prentice Hall International.

4 Wegner, P (1987). "Dimensions of Object-Based Language Design". Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87). Special Issue of ACM SIGPLAN Notices, vol. 22, pp. 168-82.