Shojiro Nishio   Akinori Yonezawa (Eds.)

Object Technologies for Advanced Software

First JSSST International Symposium
Kanazawa, Japan, November 4-6, 1993
Proceedings

Springer-Verlag
Berlin Heidelberg New York
London Paris Tokyo
Hong Kong Barcelona
Budapest

Series Editors

Gerhard Goos
Universität Karlsruhe
Postfach 69 80
Vincenz-Priessnitz-Straße 1
D-76131 Karlsruhe, Germany

Juris Hartmanis
Cornell University
Department of Computer Science
4130 Upson Hall
Ithaca, NY 14853, USA

Volume Editors

Shojiro Nishio
Department of Information Engineering, Osaka University
Suita, Osaka 565, Japan

Akinori Yonezawa
Department of Information Science, University of Tokyo
Hongo Bunkyo-Ku, Tokyo 113, Japan

CR Subject Classification (1991): D.1-4, H.2

ISBN 3-540-57342-9 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-57342-9 Springer-Verlag New York Berlin Heidelberg

Typesetting: Camera-ready by author
45/3140-543210 - Printed on acid-free paper

Preface

This volume is the Proceedings of the International Symposium on Object Technologies for Advanced Software (ISOTAS). Currently object technologies are attracting much attention in diverse areas of research and development for advanced software. Object-oriented programming holds great promise in reducing the complexity of large scale software development, and recent research in this field opens up new paradigms for parallel and reflective computing. Object-oriented databases are expected to serve as a model for next-generation database systems, by overcoming the limitations of conventional data models. Furthermore, recent research in software object bases is aimed at developing a uniform approach to the management of software artifacts produced in the software development process, such as specifications, manuals, programs, and test data, which traditionally were managed in a very ad hoc and arbitrary manner.

Active research and experimentation on object technologies in these diverse areas suggest that there are some underlying, fundamental principles common to a wide range of software development activities. The first of the JSSST (Japanese Society for Software Science and Technology) international series of symposia focuses on this topic. The aim of this symposium is to bring together leading researchers in the areas of object-oriented programming, object-oriented databases, and software object bases. We hope to promote an understanding of object technologies in a wider context and to make progress towards the goal of finding better frameworks for future advanced software development.

The Program Committee received 92 submissions from 18 different countries in Europe, America, Asia, and Australia (including 31 domestic submissions). Each submission was reviewed by at least three members of the Program Committee and sometimes by external referees. This volume contains 25 contributed papers and 6 invited papers presented at the symposium. The contributed papers were selected by a highly competitive process, based on referee reports and painstaking deliberations by members of the Program Committee.

We would like to thank all the people who made the symposium possible, including the object technology researchers who submit their works to this symposium and all those who contributed their expertise and time in reviewing the submissions.

August 1993

Shojiro Nishio, Akinori Yonezawa
Co-Chairs, Program Committee

(Invited Paper)

Uniting Functional and Object-Oriented Programming

John Sargeant[*]

Department of Computer Science
University of Manchester
Manchester M13 9PL
Tel: +44 61 275 6292
Fax: +44 61 275 6236
js@cs.man.ac.uk

Abstract

United Functions and Objects (UFO) is a general-purpose, implicitly parallel language designed to allow a wide range of applications to be efficiently implemented on a wide range of parallel machines while minimising the conceptual difficulties for the programmer. To achieve this, it draws on the experience gained in the functional and object-oriented "worlds" and attempts to bring these worlds together in a harmonious fashion.

Most of this paper concentrates on examples which illustrate how functions and objects can indeed work together effectively. At the end, a number of issues raised by early experience with the language are discussed.

1 Introduction

Modern computers are parallel. Most programming languages assume they are serial. There is an obvious need to advance beyond data parallelism and threads packages (useful though those are). However, the various forms of implicit parallelism explored during the 80s (functional, and/or parallel logic, concurrent object-oriented etc. [McG+85, Nik88, Agha86, Yon90, Am87, UeCh90]) have, by and large, made little impact on real use of parallel machines. One problem has been lack of convincing demonstrations of performance (although this is changing - see below). An equally important reason, in the author's view, is that many such languages have been too narrowly focussed, and have not incorporated the best of modern programming language technology.

UFO is not a narrow "single paradigm" language. It has been influenced by a number of disparate language styles, leading to an interesting, and potentially very useful, synthesis. The main influences are as follows:

1.1 Dataflow languages, especially SISAL

SISAL [McG+85] is a pure functional language with strict semantics, primarily geared towards numerical computation. It can be classed as a dataflow language, in that the

[*]During the early work on UFO, the author was supported directly by the Department of Computer Science. Recent work has been funded by Science and Engineering Research Council grant GR/J 11089.

underlying computational model is a parallel dataflow one. Sequencing is by data dependence only; parallelism is the default.

A great deal of work has been done on optimising SISAL for conventional supercomputers, and recently substantial numeric SISAL programs have been shown to run faster on multiprocessor Crays than Fortran versions [Cann92]. Functional languages need not be inefficient, at least for such applications. However, SISAL is quite a limited language (e.g. it has no polymorphism, data abstraction, or higher-order features; multidimensional arrays have to be represented as arrays of arrays). An update, SISAL2, has been defined [Bo+91] which addresses some, but by no means all, of these limitations.

The original idea behind UFO was to create a language based on SISAL which could be used for a wider range of parallel applications, by adding objects to encapsulate updateable state. In fact, UFO has gone well beyond this original idea, but still has a subset which (apart from syntactic differences) is very similar to SISAL.

1.2 Object-oriented languages, especially Eiffel

Issues of software reliability, reuse etc. are even more critical in parallel programming than in the old sequential world. It rapidly became clear that UFO must have good encapsulation and abstraction mechanisms, and a flexible static type system. A survey of object-oriented languages rapidly showed that Eiffel [Mey88, Mey92] was closest to what we were looking for. In particular, the Eiffel type system, with its elegant combination of genericity and inheritance looked like a good starting point.[1]

The UFO type system is therefore heavily influenced by Eiffel, although currently the rules for redefinition on inheritance are more restrictive, in order to avoid complex global validity checking. Unfortunately, to someone brought up in the functional/dataflow world, Eiffel looks extremely imperative and serial, and so the runtime semantics of UFO are very different.

1.3 Pure lazy functional languages, notably Haskell

Modern pure functional languages, for which Haskell [Hud+91] is now the standard, are characterised by higher-order functions, lazy evaluation, and strong static type systems with marked similarities to that of Eiffel.

Initially, UFO allowed constant function values, but not full higher-order functions, as it was feared that the latter would over-complicate the type system. Early experience showed that a partial parameterisation mechanism was useful, and gave no particular problems. Examples appear below.

Lazy evaluation, however, was never an option. Although it improves the expressiveness of a pure functional language, lazy evaluation is incompatible with the presence of updateable variables, as the execution order is almost impossible to visualise. Furthermore, the semantics require normal order evaluation, which is sequential. To exploit parallelism in a lazy language, it is necessary to do strictness analysis, in much the same way as it is necessary to do dataflow analysis to extract parallelism from an imperative language. It is still unclear how successfully this can be done. The effect of lazy evaluation can be simulated in UFO for those applications which really benefit from it.

[1]The author is firmly of the opinion that static typing is a Good Thing, except for a few specialised applications. By definition, the end user never sees static type errors, but may well see runtime ones! This prejudice has been strengthened by early experience with UFO, as explained in section 6.2.

A further difficulty with laziness, at least in its most general form, is that it conflicts with dynamic binding. In order to dynamically bind on an object (i.e. the first argument to a function), it is necessary to evaluate it. Few programmers outside the pure FP community are likely to regard laziness as a higher priority than dynamic binding. Compromise solutions are possible, such as using lenient, rather than lazy evaluation, or restricting laziness to certain data structures, although such compromises are rather against the spirit of pure lazy FP.

Another interesting aspect of Haskell is its type classes, which are a systematic way of dealing with overloading, and a first step towards "proper" classes and inheritance. For instance, there is a type class `Eq` which includes all types with equality defined, and it has a subclass `Ord` of types which also define ordering. Type classes can be used to impose constraints on generic types.[2] For instance, the type of a sorting function is:

```
sort :: Ord a => [a] -> [a]
```

"sort is of type list of a to list of a provided a is an instance of `Ord`". The ordering operators (< etc.) can then be used within `sort` in the knowledge that any actual type provided will have implementations of them.

However, the actual implementation of the operations is concentrated in the instances (actual types) at the leaves of the class hierarchy. It is not possible to inherit from an instance, and so the normal OO practice of incrementally adding implementation down the hierarchy is not possible. This is very restrictive and UFO has a more conventional inheritance mechanism.

1.4 Concurrent object-based languages, particularly ABCL

In concurrent object-based languages, such as early actor languages [Agha86], POOL [Am87], ABCL [Yon90], and HAL[HoAg92], a computation is expressed as a network of communicating objects, each of which manages its own local state. The design of UFO was particularly influenced by ABCL.

There are considerable similarities between UFO (stateful) objects and ABCL objects; they provide mutual exclusion on method accesses, so ensuring coherent updating of the instance variables. Incoming messages/method calls are queued if necessary. An object may continue to execute a method after it has returned a result, and may in some circumstances accept another message before it has returned a result. An object may selectively accept some messages and not others.

However, the differences are substantial; the model underlying ABCL is one of communicating sequential threads. As a result there is a distinction between different sorts of message passing ("past", "present" and "future") which is unnecessary in UFO. ABCL also has a notion of pre-emption ("express messages") which relies on the existence of such threads, and seems inappropriate (and hard to implement) for UFO.

More recently, it has become clear that there are interesting similarities between HAL[HoAg92] and UFO. Unlike earlier actor languages, HAL does have inheritance, and it also has a single-assignment update scheme very similar to that of UFO.[3]

[2]Readers familiar with Eiffel will notice a similarity to constrained generics.

[3]The traditional actors' primitive "become" is rather different from that described below, as it updates the whole state at once.