Object-Oriented Reflection and Metalevel Architectures

ECOOP '92 Workshop W5
Utrecht, the Netherlands
30 June 1992, Tuesday

PAPERS

Frank Buschmann, Konrad Kiefer, Frances Paulish, and Michael Stal
Siemens AG *mop@ztivax.siemens.com*
A Runtime Type Information System for C++

Harley Davis Ilog, S. A. *davis@ilog.ilog.fr*
Reflectively Optimizing Generic Function Dispatch

Paul Dourish Rank Xerox EuroPARC *dourish@europarc.xerox.com*
Applying Reflection for CSCW Design

Anton Eliens Vrieje Univ. Amsterdam *eliens@cs.vu.nl*
Metaprogramming in DLP

Brian Foote University of Illinois *foote@cs.uiuc.edu*
Objects, Reflection, and Open Languages

Yuuji Ichisugi, Satoshi Matsuoka, and Akinori Yonezawa
University of Tokyo *{ichisugi,matsu,yonezawa}@is.s.u-tokyo.ac.jp*
RbCl: A Reflective Concurrent Language without a Run-time Kernel

Donald Liib RIT, Stockholm *donald@tds@kth.se*
A Note on Communicational Reflection

Theo de Ridder Prometa, bv. *ridder@serc.nl*
Applying Reflection to Distributed Administative Environments

Patrick Steyaert Free University Brussels *prsteyae@vnet3.vub.ac.be*
Towards a Calculus for Objects and its Reflective Variant

Josef Templ ETH, Zurich *templ@inf.ethz.ch*
Reflection in Oberon

Handong Wu TDS, KTH *wu@tds.kth.se*
A Note on the Implications and Implementation of Reflection

Workshop Chair
Brian Foote University of Illinois at Urbana-Champaign *foote@cs.uiuc.edu*

Workshop Committee
Pierre Cointe Ecole des Mines de Nantes
Satoshi Matsuoka University of Tokyo
Workshop Advisors
Gregor Kiczales Xerox PARC
Mamdouh Ibrahim EDS/Artificial Intelligence Services

PROGRAM

Introduction, Reflective Principles (Moderator: Foote)
B. Foote, P. Steyaert
Reflection in Static Languages (Moderator: Matsuoka)
J. Templ, F. Buschmann
Reflection in Dynamic Languages (Moderator: Cointe)
H. Davis, B. Foote
Reflection and Distributed Programming (Moderator: Matsuoka)
A. Eliens, D. Liib
Reflection and Concurrency (Moderator: Cointe)
H. Wu, S. Matsuoka
Reflection and Distributed Applications (Moderator: Foote)
T. de Ridder, P. Dourish
Concluding Remarks

SUMMARY

The workshop opened with each participant giving their answer to the question: What is object-oriented reflection? A typical answer follows. Its is noteworthy that all the answers given were pretty much consistent with this one.

An object-oriented reflective system is one that is itself built out of programmable, first-class objects. A running program may dynamically inspect or alter, or otherwise manipulate such objects as it sees fit.

The program consisted of 15 minute presentations of each paper, followed by general discussions. During these discussions, the following general themes emerged:

The work on the MIP, and other work with static languages such as C++, demonstates that programmers and designers are confronting limitations that are the result of a lack of reflective facilities in these languages. The fact that reflective features are being reinvented by so many different organizations in various ad-hoc ways is evidence of their utility. A number of participants seemed gratified that the need for these sorts of facilities has come to be recognized by the broader object-oriented community.

Reflection, as valuable as it may be for conventional serial applications, will really come into its own as a way of coping with concurrency. Since the metalevel is a richer, more interesting, more complicated place in concurrent and distributed applications, the range of useful, even essential customizations and interventions is correspondingly greater.

Building entire systems out of first-class, dynamic objects can lead to a reevaluation of the relationships among system components such as compilers, operating systems, processes, and databases, and how objects may dynamically migrate among these.

ABSTRACTS

The remainder of this section contains abstracts or abridged introductions taken from the workshop position papers submitted by the participants. The full position papers can be gotten via anonymous ftp from **p300.cpl.uiuc.edu** (128.174.72.1) from the directory *pub/utrecht*. Some of the work described herein has appeared, or shall appear, elsewhere. Interested readers are invited to contact the authors of these papers directly for additional information.

A Runtime Type Information System for C++
Frank Buschmann, Konrad Kiefer, Frances Paulish, and Michael Stal
Siemens AG

In the area of C++ programming a sophisticated mechanism for runtime type identification and access to runtime type information is required to supportmanny applications and libraries. This paper describes an adequate runtime information system for the C++ programming language: the **Meta-Information Protocol** (MIP). An integration strategy is presented to demonstrate how type information can be generated and how the MIP can be included and used within C++ applications. The MIP itself is implemented in a library class hierarchy with a base class called **Meta**. It includes member functions to access type information about classes, data, and function members of classes and their types. The MIP is based on the object-oriented approach. It supports typesafe downcast for the C++ language. In addition, development of libraries and toolkits is supported as well as integration of different libaries.

Reflectively Optimizing Generic Function Dispatch

Harley Davis Ilog, S. A.

In ΤΕΛΟΣ and similar languages, a *generic function* is a function whose executed code depends on the class of its arguments. Each piece of executed code is represented by a *method*, which stores, among other things, a list of argument classes for which the method is applicable, and the code to be executed. Abstractly, when a generic function is called, it must determine the complete list of all methods applicable for the arguments, sorted form the most to least specific. The first is called, the others are called sequentially by calls to **call-next-method** within a method body.

For generic functions to compete with normal functions, this process must be optimized as much as possible. This paper presents several generic function call optimization techniques based on reflective examination of the generic function's environment. By reflective, we simply mean that the relevant aspects of the environment are explicitly represented and that a standard protocol exists to examine them. None of these techniques require special compiler support. Most of them are implemented in LE-LISP version 16, a commerical product solod by Ilog, S.A.

Applying Reflection for CSCW Design

Paul Dourish Rank Xerox EuroPARC

This paper outlies work in progress concerning the application of computational reflection to the design of systems for Computer-Supported Cooperative Work (CSCW). A metalevel architecture for the design of a CSCW toolkit provides a novel approach to adquate provision of flexibility in the design of these systems. In particular, it points towards the ability to support dynamic flexibility in group behaviour, which is critical to the usability of such systems.

Metaprogramming in DLP

Anton Eliens Vrieje Universiteit, Amsterdam

Prolog offers powerful features for metaprogramming. These features may be used for extending the language to incorporate other programming paradigms, such as, for example, the object-oriented paradigm, or to embed special purpose inference procedures in the language. However, the penalty of using meta-programming to extent the language in this way is an obvious loss of efficiency.

We have developed the distributed logic programming language DLP, combining logic programming with object-oriented features and parallelism. Our approach allows for an efficient implementation since the computational model of DLP directly supports the features derived from its underlying paradigms. Nevertheless, the language DLP supports metaprogramming in the style of Prolog. For example, metaprogramming may be used to implement a restricted form of and-parallelism.

Concluding our paper, well will discuss the possible ways to include reflective capabilities by extending our (prototype-based) object model to one based on a class/metaclass concept. Our interest is motivated by our

current effort to reimplement DLP in Active C++, an extendsion of C++ that supports active classes and communication by rendevous.

Objects, Reflection, and Open Languages

Brian Foote University of Illinois at Urbana-Champaign

The marriage of reflection with object-oriented programming and design techniques holds out the promise of dramatically changing the way that we think about, organize, implement, and use programming languages and systems. The combination of object-oriented programming languages and reflective metalevel architectures allows the full power of the object-oriented approach to be brought to bear upon object-oriented languages themselves. Together, they have the potential to permit the long deferred promise of truly open programming languages and systems to be realized.

RbCl: A Reflective Concurrent Language without a Run-time Kernel

Yuuji Ichisugi, Satoshi Matsuoka, and Akinori Yonezawa
University of Tokyo

We propose a reflective object-oriented concurrent language RbCl which has no run-time kernel. That is to say, all behaviors of RbCl except for what is restricted by the operating system and hardware can be modified/extended by the user. RbCl runs efficiently in a distributed environment and is intended for practical use. The execution of an RbCl program is performed by a *metasystem* that consists of *metalevel objects*. All the features of RbCl including concurrent execution, inter-node communication, and even reflective facilities themselves are realized by the metalevel objects. Important metalevel objects are called *system objects*, that are registered in *system object tables*. The user can change the behavior of the metasystem by replacing elements of the system object tables with user-defined objects. RbCl also has a novel feature called *linguistic symbiosis* with the metalevel objects. All the metalevel objects in the initial RbCl metasystem are actually C++ objects, but linguistic symbiosis enables the user to manipulate metalevel C++ objects just as ordinary RbCl objects. Even reflective schemes and facilities themselves are realized by system objects that can be modified/extended by the user. Therefore, debugging of reflective programs and experiments on reflective schemes and facilities can be expressed and performed within the RbCl language framework.

A Note on Communicational Reflection

Donald Liib The Royal Institute of Technology, Stockholm

Two reflective aspects of object-based programming can be defined in the following ways: A system is *computationally* reflective if can reason about and act upon itself. A system is *communicatively* reflective if it has the ability to chose an order of computations that guarentees reasonable efficiency to reflective computations.

One way to achieve this is to introduce flowing visibility. The range of visibility can be defined from fully public to fully private.

Applying Reflection to Distributed Administative Environments

Theo de Ridder Prometa, bv.

In traditional administrative systems the emphasis is on the data independence of large amounts of data. Data independence is typically obtained by using relatively simple data models and centralised databases. But, in spite of simple data models, real information systems appeared hard to build, modify, and maintain. Many of the problems of information engineering are due to scale: huge amounts of data, a wide range of involved people, a complex surrounding world, and massive mainframes.

We discuss a reflective data-oriented approach to enrich existing administrative systems with extensible object capabilities for communication is distributed environments. The kernel of our solution is a samll

interpretive engine (ALOM) that handles symbolic data structures, containing passive and active data. The first implementation of ALOM is implemented in C and incorporated as externals in HyperCard.

Towards a Calculus for Objects and its Reflective Variant

Patrick Steyaert Free University Brussels

First, a calculus is constructed that models, in a direct way, the crucial features of object-oriented programming. Then, we propose an extension with behavioural and structural reflection. Behavioural and structural reflection are added independently of each other, but support each other. The semantics of both the calculus and its extension are made clear by examples.

Behavioral reflection is achieved by absorption and reification of the evaluation method typical for expressions (respectively by quoting and its inverse, unquoting, operation). Evaluation is done in an explicitly given context (as well as for reified and absorbed evaluation methods), paying attention to make a distinction between object-level context and metalevel context. This leads in a natural way to the idea of having a stack of contexts (object-level meta(1)-level, meta(2)-level...).

Structural reflectionis achieved by absorption and reification of the send method typical for objects. Special care is taken that one does not have to breach the encapsulation, that is inherent in the non-reflective calculus, in order to make use of structural reflection, an aspect that is ignored in most other work.

Reflection in Oberon

Josef Templ ETH, Zurich

As part of an attempt to make extensible operating systems more powerful, we investigated the benefits and costs that may result from giving more attention to reflection. As a case study, we attempted to introduce a reflective architecture into the Oberon system. Since Oberon is intended to be a general purpose programming system, we strove to make the resulting system efficient. It is claimed that the benefits of a reflective architecture will only be embraced by a larger community if this architecture can be efficiently implemented. Thus, we focus on those reflective faciltities that can be implemented without affecting the efficiency of non-reflective computation.

A Note on the Implications and Implementation of Reflection

Handong Wu TDS, KTH

It is claimed that concurrent object-oriented programming models (COOPS), such as the *actor* model and its derivatives, would help make it easier to write good parallel programs, through combining the computational power of concurrency with the modeling power of sequential object-oriented programming. Objects provide an effective means for structuring sequential software systems in terms of components and interfaces, as well as a useful abstraction for writing parallel programs. An object can maintain its own internal parallelism and control concurrency constraints imposed on it by other objects.

However, this claim cannot be fulfilled for distributed real-time applications. In this paper, we will discuss some implications of reflection on describing predictable and portable distributed realtime applications, as well as how to efficiently implement a reflective system.