LNCS 4228

David E. Lightfoot
Clemens A. Szyperski (Eds.)

Modular Programming Languages

7th Joint Modular Languages Conference, JMLC 2006
Oxford, UK, September 2006
Proceedings

Springer

David E. Lightfoot   Clemens A. Szyperski (Eds.)

Modular Programming Languages

7th Joint Modular Languages Conference, JMLC 2006
Oxford, UK, September 13-15, 2006
Proceedings

Springer

Volume Editors

David E. Lightfoot
Oxford Brookes University
School of Technology
Department of Computing
Oxford, OX33 1HX, UK
E-mail: dlightfoot@brookes.ac.uk

Clemens A. Szyperski
One Microsoft Way
Redmond WA 98052, USA
E-mail: cszypers@microsoft.com

Library of Congress Control Number: 2006932028

CR Subject Classification (1998): D.3, D.2, D.1, D.4, F.3

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN	0302-9743
ISBN-10	3-540-40927-0 Springer Berlin Heidelberg New York
ISBN-13	978-3-540-40927-4 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11860990 06/3142 5 4 3 2 1 0

Preface

On behalf of the Steering Committee we are pleased to present the proceedings of the 2006 Joint Modular Languages Conference (JMLC), organized by Oxford Brookes University, Oxford, UK and held at Jesus College, Oxford. The mission of JMLC is to explore the concepts of well-structured programming languages and software and those of teaching good design and programming style. JMLC 2006 was the seventh in a series of successful conferences with themes including the construction of large and distributed software systems, and software engineering aspects in new and dynamic application areas.

We were fortunate to have a dedicated Program Committee comprising 41 internationally recognized researchers and industrial practitioners. We received 36 submissions and each paper was reviewed by at least three Program Committee members (four for papers with an author on the Program Committee). The entire reviewing process was supported by the OpenConf system. In total, 23 submissions were accepted along with two invited papers and are included in this proceedings volume.

For the successful local organization of JMLC we thank Muneera Masterson, Ali McNiffe and Fiona Parker and other staff and student helpers of Oxford Brookes University as well as Rosemary Frame and Jo Knighton and other staff of Jesus College, Oxford. The proceedings you now hold were published by Springer and we are grateful for their support. Finally, we must thank the many authors who contributed the high-quality papers contained within these proceedings.

September 2006

David Lightfoot
Clemens Szyperski

Organization

JMLC 2006 was the seventh conference in a successful series, with past events held in:

1987 in Bled, Slovenia;
1990 in Loughborough, UK;
1994 in Ulm, Germany;
1997 in Linz, Austria;
2000 in Zürich, Switzerland;
2003 in Klagenfurt, Austria.

Steering Committee

László Böszörményi, University of Klagenfurt, Austria
Michael Franz, UC Irvine, USA
Jürg Gutknecht, ETH Zürich, Switzerland
David Lightfoot, Oxford Brookes University, UK (Program Co-chair and Local Organizer)
Hanspeter Mössenböck, University of Linz, Austria
Clemens Szyperski, Microsoft, USA (Program Co-chair)
Niklaus Wirth, ETH Zürich emeritus, Switzerland

Program Committee

Jonathan Aldrich, CMU, USA
Pierre America, Philips Research, Netherlands
Uwe Assmann, TU Dresden, Germany
Nick Benton, Microsoft Research Cambridge, UK
László Böszörményi, University of Klagenfurt, Austria
Gilad Bracha, Sun Java Software, USA
Michael E. Caspersen, Aarhus University, Denmark
Craig Chambers, University of Washington, USA
Michael Franz, UC Irvine, USA
K. John Gough, Queensland UT, Australia
Dominik Gruntz, Fachhochschule Aargau, Switzerland
Jürg Gutknecht, ETH Zürich, Switzerland
Thomas Henzinger, EPF Lausanne, Switzerland
Nigel Horspool, University of Victoria, Canada
Zoltán Horváth, Budapest University (ELTE), Hungary
Mehdi Jazayeri, TU Vienna, Austria
Helmut Ketz, Fachhochschule Reutlingen, Germany

Brian Kirk, Robinson Associates, UK
Christoph Kirsch, University of Salzburg, Austria
Jens Knoop, TU Vienna, Austria
Kai Koskimies, TU Tampere, Finland
Liu Ling, University of Shanghai, China
Jochen Ludewig, University of Stuttgart, Germany
Jan Madey, University of Warsaw, Poland
Ole Lehrmann Madsen, Aarhus University, Denmark
Roland Mittermeir, University of Klagenfurt, Austria
Hanspeter Mössenböck, University of Linz, Austria
Pieter Muller, Esmertec, Switzerland
Judit Nyeky, Budapest University (ELTE), Hungary
Martin Odersky, EPF Lausanne, Switzerland
Jens Palsberg, Purdue University, USA
Frank Peschel-Gallee, Microsoft, USA
Gustav Pomberger, University of Linz, Austria
Wolfgang Pree, University of Salzburg, Austria
Paul Reed, Padded Cell Software, UK
Paul Roe, Queensland UT, Australia
Markus Schordan, TU Vienna, Austria
Brian Shearing, The Software Factory, UK
Pat Terry, Rhodes University, South Africa
Fyodor Tkachov, Institute for Nuclear Research (RAS), Russia
Wolfgang Weck, Independent Software Architect, Switzerland
Mark Woodman, Middlesex University, UK

Additional Reviewer

Christian Wimmer, University of Linz, Austria

Sponsoring Institutions

Microsoft Research
Sun Microsystems
Robinson Associates
dpunkt Verlag

Table of Contents

Separating Concerns with Domain Specific Languages

Steve Cook

Microsoft UK Ltd, Cambridge
steve.cook@microsoft.com

Abstract. I'll talk about the separation of concerns in the development of large distributed enterprise systems, how to manage it using domain specific languages, and how to build these languages. This brief note outlines some of the topics I'll cover.

1 Separation of Concerns

Most developments in programming language design are intended to improve the ability of the programmer to separate the expression of different concerns. This has progressively led to the development of language features such as procedures, abstract data types, objects, polymorphic type systems, aspects, and so on.

We're now moving into an era when the normal case of software development is distributed and heterogeneous, with the internet playing a pivotal role. It's simply not practical today to use a single programming language to create all aspects of a large and complex computing system. Different technologies are used to implement user-interfaces, business subsystems, middleware, databases, workflow systems, sensors, etc. Enterprise programming stacks include as first-class participants a variety of inter-related programming and scripting languages, databases, metadata and configuration files. Most programming projects involve interoperating with what is already there, which requires interfacing to existing technology stacks.

In such a world, concerns such as the structure and organization of business data and processes inherently span multiple technologies. A given business concept will show up in the user interface, in the formats used to communicate between components, in the interfaces offered from one component to another, in the schemas for databases where business data is stored, and in the programming logic for manipulating all of the above. Even the simplest change, such as changing the name of the business concept, impacts all of these pieces. Such concerns cannot possibly be effectively separated by improving programming language design. How then can we approach this problem?

2 Development Using Domain Specific Languages

A promising approach has been described variously as "Language-Oriented Programming" [1], "Language Workbenches" [2], "Generative Programming" [3] and "Model Driven Engineering" [4]. All of these phrases essentially describe the same pattern: given a class of problems, design a special-purpose language – a Domain Specific Language or DSL - to solve it.

A simple (and old) example of this pattern is the language of regular expressions. For example, using the .Net class System.Text.RegularExpression.Regex, the regular expression "(?<user>[^@]+)@(?<host>.+)" applied to a string of characters will find email addresses in it, and for each address found, appropriately extract the user and host variables. Programming such a procedure directly in a general-purpose language is a significantly larger and more error-prone task.

In developing complex enterprise systems, it is increasingly the case that graphical languages can be used to express certain concerns most effectively. Business workflows, business data, and system, application and data centre configuration are obvious candidates for graphical representation. Also textual languages, while effective for inputting large programs, may not be the most effective medium for displaying, analyzing and interpreting these programs.

Putting these ingredients together provides the motivation for an emerging class of graphical language-processing tools, which includes the DSL Tools from Microsoft [5], the Generic Modeling Environment (GME) from Vanderbilt University [6], and commercial examples from MetaCase, Xactium and others. These tools enable the language author to design and implement the abstract and concrete syntax for a DSL, together with the ancillaries needed to integrate the language into a development process.

Of course it is not sufficient simply to design what a DSL looks like; it is also necessary to give its expressions meaning, which in practical terms means to generate executable artifacts from it: these will most likely be programs in more general-purpose languages, together with configuration files, scripts and metadata, that can be deployed together to implement the intention of the developer.

As soon as generation is introduced into the development process, there is the possibility of developers changing the generated artifacts. Uncontrolled, this will break the process: the source form of the DSL will become out of date and useless. Alleviating this issue is one of the main challenges of making DSLs successful. Not all artifacts can be generated from DSLs, so it is essential to be able to interface generated artifacts with hand-coded ones: various language techniques such as partial classes [7] can enable this.

3 Software Factories

A DSL can provide a means to simplify the development of one area of concern. But the development of large distributed applications involves the integration of multiple areas of concern, with multiple stakeholders manipulating the system via multiple different viewpoints.

Managing the complexity of such a development involves delivering appropriate languages, tools and guidance to individual participants in the process at the right place and time. Enabling this is the province of Software Factories [8], an approach to software development that focuses on the explicit identification of viewpoints in the development process, the definition of DSLs, tools and guidance to support these viewpoints, and the delivery of these capabilities to individuals during the enactment of the process.

References

1. Dimitriev, S. Language-Oriented Programming: The Next Programming Paradigm, http://www.onboard.jetbrains.com/is1/articles/04/10/lop/
2. Fowler, M. Language Workbenches: The Killer App for Domain Specific Languages? http://martinfowler.com/articles/languageWorkbench.html
3. Czarnecki, K. and Eisenecker, U.W. Generative Programming – Methods, Tools and Applications. Addison-Wesley (2000).
4. Bézivin J., Jouault F, and Valduriez P. On the Need for Megamodels. Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (2004).
5. DSL Tools Workshop. http://msdn.microsoft.com/vstudio/DSLTools/
6. Akos Ledeczi, Miklos Maroti, Arpad Bakay, Gabor Karsai, Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J. and Volgyesi, P. The Generic Modeling Environment. Proceedings of WISP'2001, May, 2001. http://www.isis.vanderbilt.edu/Projects/gme/GME2000Overview.pdf
7. C# programming guide, http://msdn2.microsoft.com/en-us/library/wa80x488.aspx
8. Greenfield, J., Short, K., Cook, S., Kent, S. Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools. Wiley (2004).

Event-Based Programming Without Inversion of Control

Philipp Haller and Martin Odersky

École Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland

1 Introduction

Concurrent programming is indispensable. On the one hand, distributed and mobile environments naturally involve concurrency. On the other hand, there is a general trend towards multi-core processors that are capable of running multiple threads in parallel.

With *actors* there exists a computation model which is especially suited for concurrent and distributed computations [16,1]. Actors are basically concurrent processes which communicate through *asynchronous message passing*. When combined with *pattern matching* for messages, actor-based process models have been proven to be very effective, as the success of Erlang documents [3,25].

Erlang [4] is a dynamically typed functional programming language designed for programming real-time control systems. Examples of such systems are telephone exchanges, network simulators and distributed resource controllers. In these systems, large numbers of concurrent processes can be active simultaneously. Moreover, it is generally difficult to predict the number of processes and their memory requirements as they vary with time.

For the implementation of these processes, operating system threads and threads of virtual machines, such as the Java Virtual Machine [22], are usually too heavyweight. The main reasons are: (1) Over-provisioning of stacks leads to quick exhaustion of virtual address space and (2) locking mechanisms often lack suitable contention managers [12]. Therefore, Erlang implements concurrent processes by its own runtime system and not by the underlying operating system [5].

Actor abstractions as lightweight as Erlang's processes have been unavailable on popular virtual machines so far. At the same time, standard virtual machines are becoming an increasingly important platform for exactly the same domain of applications in which Erlang–because of its process model–has been so successful: Real-time control systems [23,27].

Another domain where virtual machines are expected to become ubiquitous are applications running on mobile devices, such as cellular phones or personal digital assistants [20]. Usually, these devices are exposed to severe resource constraints. On such devices, only a few hundred kilobytes of memory is available to a virtual machine and applications.

This has important consequences: (1) A virtual machine for mobile devices usually offers only a restricted subset of the services of a common virtual machine for desktop or server computers. For example, the KVM[1] has no support for reflection (introspection) and serialization. (2) Programming abstractions used by applications have to be very lightweight to be useful. Again, thread-based concurrency abstractions are too heavyweight. Furthermore, programming models have to cope with the restricted set of services a mobile virtual machine provides.

A common alternative to programming with threads are event-driven programming models. Programming in explicitly event-driven models is very difficult [21].

Most programming models support event-driven programming only through *inversion of control.* Instead of calling blocking operations (e.g. for obtaining user input), a program merely registers its interest to be resumed on certain *events* (e.g. an event signaling a pressed button, or changed contents of a text field). In the process, *event handlers* are installed in the execution environment which are called when certain events occur. The program never calls these event handlers itself. Instead, the execution environment dispatches events to the installed handlers. Thus, control over the execution of program logic is "inverted".

Virtually all approaches based on inversion of control suffer from the following two problems: First, the interactive logic of a program is fragmented across multiple event handlers (or classes, as in the state design pattern [13]). Second, control flow among handlers is expressed implicitly through manipulation of shared state [10].

To obtain very lightweight abstractions without inversion of control, we make actors *thread-less.* We introduce *event-based actors* as an implementation technique for lightweight actor abstractions on *non-cooperative* virtual machines such as the JVM. Non-cooperative means that the virtual machine provides no means to explicitly manage the execution state of a program.

The central idea is as follows: An actor that waits in a receive statement is not represented by a blocked thread but by a closure that captures the rest of the actor's computation. The closure is executed once a message is sent to the actor that matches one of the message patterns specified in the receive. The execution of the closure is "piggy-backed" on the thread of the sender. If the receiving closure terminates, control is returned to the sender as if a procedure returns. If the receiving closure blocks in a second receive, control is returned to the sender by throwing a special exception that unwinds the receiver's call stack.

A necessary condition for the scheme to work is that receivers never return normally to their enclosing actor. In other words, no code in an actor can depend on the termination or the result of a receive block. We can express this non-returning property at compile time through Scala's type system. This is not a severe restriction in practice, as programs can always be organized in a way so that the "rest of the computation" of an actor is executed from within a receive. To the best of our knowledge, event-based actors are the first to (1) allow reactive behavior to be expressed without *inversion of control*, and (2) support arbitrary blocking operations in reactions, at the same time. Our actor library outperforms other state-of-the-art actor languages with respect to message passing

[1] See http://java.sun.com/products/cldc/.

speed and memory consumption by several orders of magnitude. Our implementation is able to make use of multi-processors and multi-core processors because reactions can be executed simultaneously on multiple processors. By extending our event-based actors with a portable runtime system, we show how the essence of distributed Erlang [31] can be implemented in Scala. Our library supports virtually all primitives and built-in-functions which are introduced in the Erlang book [4]. The portability of our runtime system is established by two working prototypes based on TCP and the JXTA[2] peer-to-peer framework, respectively.

All this has been achieved without extending or changing the programming language. The event-based actor library is thus a good demonstrator of Scala's abstraction capabilities. Beginning with the upcoming release 2.1.7, it is part of the Scala standard distribution[3].

Other Related Work. Actalk [8] implements actors as a library for Smalltalk-80 by extending a minimal kernel of pure Smalltalk objects. Their implementation is not event-based and Smalltalk-80 does not support parallel execution of concurrent actors on multi-processors (or multi-core processors).

Actra [29] extends the Smalltalk/V virtual machine with an object-based real-time kernel which provides lightweight processes. In contrast, we implement lightweight actors on unmodified virtual machines.

Chrysanthakopoulos and Singh [11] discuss the design and implementation of a channel-based asynchronous messaging library. Channels can be viewed as special state-less actors which have to be instantiated to indicate the types of messages they can receive. Instead of using heavyweight operating system threads they develop their own scheduler to support continuation passing style (CPS) code. Using CLU-style iterators blocking-style code is CPS-transformed by the C# compiler.

SALSA (Simple Actor Language, System and Architecture) [30] extends Java with concurrency constructs that directly support the notion of actors. A preprocessor translates SALSA programs into Java source code which in turn is linked to a custom-built actor library. As SALSA implements actors on the JVM, it is somewhat closer related to our work than Smalltalk-based actors or channels. Moreover, performance results have been published which enables us to compare our system with SALSA, using ports of existing benchmarks.

Timber is an object-oriented and functional programming language designed for real-time embedded systems [6]. It offers message passing primitives for both synchronous and asynchronous communication between concurrent *reactive objects*. In contrast to event-based actors, reactive objects cannot call operations that might block indefinitely. Instead, they install call-back methods in the computing environment which executes these operations on behalf of them.

Frugal objects [14] (FROBs) are distributed reactive objects that communicate through typed events. FROBs are basically actors with an event-based computation model, just as our event-based actors. The goals of FROBs and

[2] See http://www.jxta.org/.

[3] Available from http://scala.epfl.ch/.