Concurrent Programming in ML

John H. Reppy

CONCURRENT PROGRAMMING IN ML

JOHN H. REPPY
Bell Labs, Lucent Technologies, Murray Hill, New Jersey

CAMBRIDGE
UNIVERSITY PRESS

PUBLISHED BY THE PRESS SYNDICATE OF THE UNIVERSITY OF CAMBRIDGE
The Pitt Building, Trumpington Street, Cambridge, United Kingdom

CAMBRIDGE UNIVERSITY PRESS
The Edinburgh Building, Cambridge CB2 2RU, UK www.cup.cam.ac.uk
40 West 20th Street, New York, NY 10011-4211, USA www.cup.org
10 Stamford Road, Oakleigh, Melbourne 3166, Australia
Ruiz de Alarcón 13, 28014 Madrid, Spain

First published 1999

Printed in the United States of America

Typefaces Times 10.75/14pt. and Courier System TEX [AU]

A catalog record for this book is available from the British Library.

Library of Congress Cataloging-in-Publication Data
Reppy, John H.
Concurrent programming in ML John H. Reppy.
p. cm.
Includes bibliographical references.
ISBN 0 521 48089 2 hardback
1. ML (Computer program language) 2. Parallel processing
(Electronic computers) I. Title.
QA76.73.M6R47 1999
005.2′752 – dc21 99-20465
CIP

ISBN 0 521 48089 2 hardback

CONCURRENT PROGRAMMING IN ML

Concurrent Programming in ML presents the language Concurrent ML (CML), which supports the union of two important programming models: concurrent programming and functional programming. CML is an extension of the functional language Standard ML (SML) and is included as part of the Standard ML of New Jersey (SML/NJ) distribution. CML supports the programming of process communication and synchronization using a unique higher-order concurrent programming mechanism which allows programmers to define their own communication and synchronization abstractions.

The main focus of the book is on the practical use of concurrency to implement naturally concurrent applications. In addition to a tutorial introduction to programming in CML, this book presents three extended examples of using CML for systems programming: a parallel software build system, a simple concurrent window manager, and an implementation of distributed tuple spaces.

This book includes a chapter on the implementation of concurrency using features provided by the SML/NJ system and provides many examples of advanced SML programming techniques. The appendices include the CML reference manual and a formal semantics of CML.

This book is aimed at programmers and professional developers who want to use CML, as well as students, faculty, and other researchers.

Preface

This book is about the union of two important paradigms in programming languages, namely, *higher-order* languages and *concurrent* languages. Higher-order programming languages, often referred to as "*functional programming*" languages,[1] are languages that support functions as first-class values. The language used here is the popular higher-order language *Standard ML* (**SML**) [MTH90, MTHM97], which is the most prominent member of the **ML** family of languages. In particular, the bulk of this book focuses on concurrent programming using the language *Concurrent ML* (**CML**), which extends **SML** with independent processes and higher-order communication and synchronization primitives. The power of **CML** is that a wide range of communication and synchronization abstractions can be programmed using a small collection of primitives.

A concurrent program is composed from two or more sequential programs, called *processes*, that execute (at least conceptually) in parallel. The sequential part of the execution of these processes is independent, but they also must interact via shared resources in order to collaborate on achieving their common purpose. In this book, we are concerned with the situation in which the concurrency and process interaction are explicit. This is in contrast with implicitly parallel languages, such as parallel functional languages [Hud89, Nik91, PvE93] and concurrent logic programming languages [Sha89]. The choice of language mechanisms used for process interaction is the key issue in concurrent programming language design. In this aspect, **CML** takes the unique approach of supporting *higher-order concurrent programming*, in which the communication and synchronization operations are first-class values, in much the same way that functions are first-class values in higher-order languages.

Concurrent programming is an especially important technique in the construction of systems software. Such software must deal with the unpredictable sequencing of external events, often from multiple sources, which is difficult to manage in sequential languages.

[1] I choose the term "higher-order" to avoid confusion with "pure" (*i.e.*, referentially transparent) functional languages.

Structuring a program as multiple threads of control, one for each external agent or event, greatly improves the modularity and manageability of the program. Concurrent programming replaces the artificial total ordering of execution imposed by sequential languages by a more natural partial ordering. The resulting program is nondeterministic, but this is necessary to deal with a nondeterministic external world efficiently.

This book differs from most books on concurrent programming in that the underlying sequential language, **SML**, is a higher-order language. The use of **SML** as the sequential sub-language has a number of advantages. **SML** programs tend to be "mostly-functional" and typically do not rely on heavy use of global state; this reduces the effort needed to migrate from a sequential to a concurrent programming style. The high-level features of **SML**, such as datatypes, pattern matching, the module system, and garbage collection, provide a more concise programming notation. Recent advances in implementation technology allow us to take advantage of the benefits of **SML**, without sacrificing good performance [SA94, Sha94, TMC^{+}96]. One of the theses of this book is that efficient system software can be written in a language such as **SML**.

History

The language design ideas presented in this book date back to the language **PML** [Rep88], an **ML** dialect developed at AT&T Bell Laboratories as part of the **Pegasus** system [RG86, GR92]. The purpose of the **Pegasus** project was to provide a better foundation for building interactive systems than that provided by the **C**/UNIX world circa 1985. We believed then, and still do, that interactive applications are inherently concurrent, and that they should be programmed in a concurrent language. This was the motivation for designing a concurrent programming language. We finished an implementation of the **Pegasus** run-time system before the design of **PML** was complete. We tested our ideas on this run-time system by writing prototype applications in **C** with calls to our concurrency library. Our experience with these applications convinced us that the concurrency features of **PML** should be designed to support abstraction. It was this design goal that led me to develop "first-class synchronous operations" [Rep88].

Shortly thereafter, I began a graduate program at Cornell University, and started working with early versions of Standard ML of New Jersey (**SML/NJ**) [AM87, AM91]. In the spring of 1989, Appel and Jim developed a new back-end for **SML/NJ**, based on a *continuation-passing style* representation [AJ89, App92]. A key feature of this back-end is that the program stack was replaced by heap-allocated return closures. In the fall of 1989, this led to the addition of first-class continuations as a language extension in **SML/NJ** [DHM91], which made it possible to implement concurrency primitives directly in **SML**. Exploiting this feature, I implemented a coroutine version of the **PML** primitives on top of **SML/NJ** [Rep89]. Others also exploited the first-class continuations provided by **SML/NJ**: Ramsey, at Princeton, implemented **PML**-like primitives [Ram90], and Cooper and Morrisett, at Carnegie-Mellon, implemented **Modula 2+** style shared-memory primitives [CM90]. Morrisett and Tolmach later implemented a multiprocessor version of low-level shared-memory primitives [MT93].

While first-class continuations provided an important mechanism for implementing concurrency primitives, they did not provide a mechanism for preemptive scheduling, which is key to supporting modular concurrent programming. To address this problem, I added support for UNIX style signal handling to the **SML/NJ** run-time system [Rep90]. With this support, I modified my coroutine version of the **PML** primitives to include preemptive scheduling, and the first version of **CML** was born. It was released in November of 1990. This implementation evolved into the version of **CML** that was described in the first published paper about **CML** [Rep91a], and was the subject of my doctoral dissertation [Rep92]. In February of 1993, version 0.9.8 of **CML** was released as part of the **SML/NJ** distribution. After that release, a major effort was undertaken to redesign the Basis Library provided by **SML** implementations [GR99]. This effort grew into what is now known as *Standard ML 1997* (**SML**'97), which includes the new basis library, as well as a number of language improvements and simplifications. From the programmer's perspective, the most notable of these changes is the elimination of imperative type variables and the introduction of new primitive types for characters and machine words [MTHM97]. **CML** has also been overhauled to be compatible with **SML**'97 and the new Basis Library, and to use more uniform naming conventions. Although some of the names have changed since version 0.9.8, the core features and concepts are the same. Most recently, Riccardo Pucella has ported **CML** to run on Microsoft's Windows NT operating system.

Since its introduction, **CML** has been used by many people around the world. Uses include experimental telephony software [FO93], as a target language for a concurrent constraint programming language [Pel92], as a basis for distributed programming [Kru93], and for programming dataflow networks [Čub94b, Čub94a]. My own use of **CML** has focused on the original motivation of the **Pegasus** work: providing a foundation for user interface construction. Emden Gansner and I have constructed a multithreaded **X Window System** toolkit, called **eXene**, which is implemented entirely in **CML** [GR91, GR93].

CML has also been the focus of a fair bit of theoretical work. The semantics of the **PML** subset of the language has been formalized in several different ways [BMT92, MM94, FHJ96]. My dissertation also presents a full semantics of the **CML** concurrency mechanisms [Rep91b, Rep92] (Appendix B presents this semantics, but without the proofs). Nielson and Nielson have worked on analyzing the communication patterns (or *topology*) of **CML** programs [NN93, NN94] as well as on control-flow analysis for **CML** [GNN97]. Such analysis can be used to specialize communication operations to provide better performance.

While **CML** is not likely to change, there will continue to be improvements and enhancements to its implementation. The most important improvement is to provide the benefits of kernel-level threads to **CML** applications (*e.g.*, to mask the latency of system calls). To provide these benefits requires a new run-time system, which is under construction as of this writing. This new run-time system should also make a multiprocessor implementation of **CML** possible. The other major effort is to build useful libraries, particularly in the area of distributed programming and network applications. The **CML** home page (see below) will provide information about these, and other, improvements as they become available.

Getting the software

The **SML/NJ** system, **CML**, **eXene**, and other related software are all available, free of charge, on the internet.

Information about the latest and greatest version of **CML**, as well as user documentation, technical papers, and the sample code from this book can be found at the *Concurrent ML* home page:

```
http://www.cs.bell-labs.com/~jhr/sml/cml/index.html
```

CML is also available as part of the **SML/NJ** distribution, which can be found at the **SML/NJ** home page:

```
http://cm.bell-labs.com/cm/cs/what/smlnj/index.html
```

This page also provides links to the **SML/NJ** Library documentation and to the online version of the *Standard ML Basis Library* manual.

Overview of the book

This book was written with several purposes in mind. The primary purpose of this book is to promote the use of **CML** as a concurrent language; it provides not only a tutorial introduction to the language, but also examples of more advanced uses. Although it is not designed as a teaching text, this book does provide an introduction to Concurrent Programming, and drafts of it have been used in courses at various universities. Because of the strong typing of **SML** and the choice of concurrency primitives, **CML** provides a friendlier introduction to concurrent programming than in many other languages. **CML** also provides a good example of systems programming using **SML/NJ**, and I hope that this book will inspire other non-traditional uses of the language. This book does not make an attempt to introduce or describe **SML**, as there are a number of books and technical reports that already fill that purpose; a list of these can be found in the Chapter 1 notes (and on the **SML/NJ** home page).

The book is loosely organized into three parts: an introduction to concurrent programming, an expository description of **CML** (essentially a **CML** tutorial), and finally, a practicum consisting of example applications.

The first chapter motivates the rest of the book by arguing the merits of concurrent programming. Chapter 2 introduces various concepts and issues in concurrent programming and concurrent programming languages.

The next four chapters focus on the design and use of **CML**. First, Chapters 3 and 4 give a tutorial introduction to the basic **CML** features and programming techniques. Chapter 5 expands on this discussion by exploring various synchronization and communication abstractions. Finally, Chapter 6 describes the rationale for the design of **CML**; this chapter is mainly intended for those interested in language design issues, and may be skipped by the casual reader.

The subsequent three chapters present extended examples of **CML** programs. While space restrictions constrain the scope of these examples, each is a representative of a natural application area for concurrent programming. Furthermore, they provide examples of complete **CML** programs, rather than just program fragments. Chapter 7 describes a controller for a simple parallel software-build system. This illustrates the use of concurrency to manage parallel system-level processes. The next example, in Chapter 8, is a toy concurrent window manager, which illustrates the use of concurrency in user interface software. Chapter 9 describes an implementation of distributed tuple spaces. This provides both an illustration of how a distributed systems interface might fit into the **CML** framework, and how systems programming can be done in **SML** and **CML**.

The book concludes with a chapter on the implementation of concurrency in **SML/NJ** using its *first-class continuations*. **SML/NJ** provides a fairly unique test-bed for experimenting in concurrent language design, and this chapter provides a "how-to" guide for such experimentation.

There are two appendices, which provide a more concise description of **CML**. Appendix A is an abridged version of the *CML Reference Manual*; the complete manual is available from the **CML** home page. Appendix B gives an operational semantics for the concurrency features of **CML**, along with statements of some of its properties (proofs can be found in my dissertation [Rep92]).

Citations and a discussion of related work are collected in "**Notes**" sections at the end of each chapter. These notes also provide some historical context. The text is illustrated with numerous examples; the source code for most of these is available from the **CML** home page.

Acknowledgements

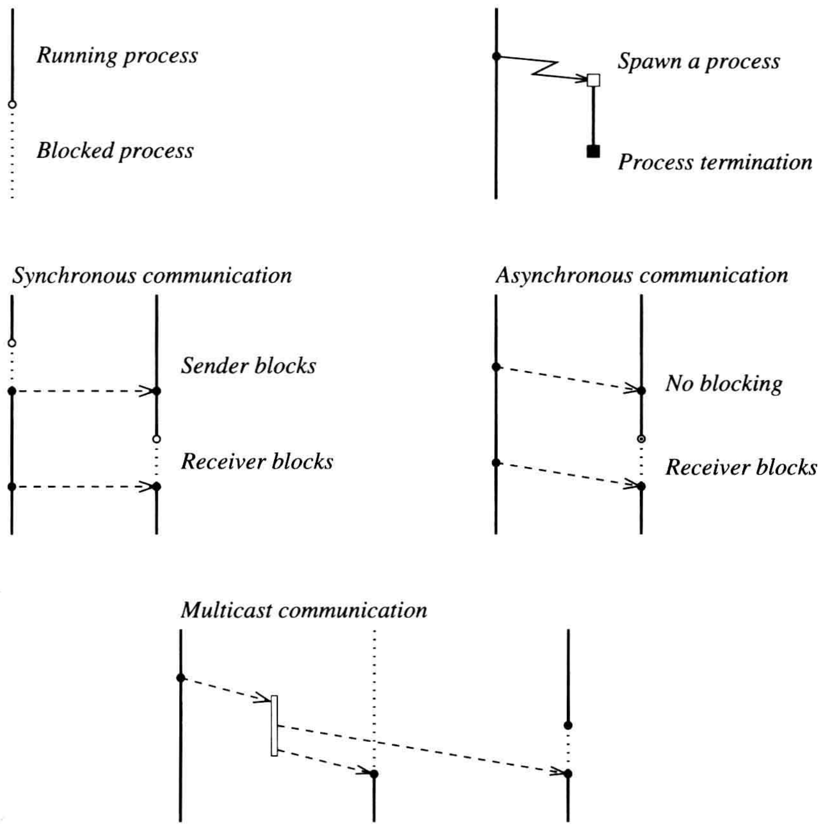
This book has taken a long time for me to write, and many people have helped along the way. Foremost, I would like to thank my wife, book coach, and primary proofreader, Anne Rogers. Without her encouragement, I doubt this book would have been finished. My graduate advisor at Cornell University, Tim Teitelbaum, provided support for the research that produced **CML**.[2] Andrew Appel and Dave MacQueen encouraged me to start this project, and have helped with implementation issues.

In addition to Anne, a number of other people provided feedback on portions of the text. Emden Gansner and Lorenz Huelsbergen performed the second pass of proofreading on many of the chapters. Greg Morrisett provided detailed feedback about Chapter 10, and Jon Riecke helped with the semantics in Appendix B. Nick Afshartous, Lal George, Prakash Panangaden, and Chris Stone also provided feedback on various parts of the book. Riccardo Pucella has helped with recent versions of the **CML** implementation, including the Windows NT port. I would also like to thank my editor at Cambridge University Press, Lauren Cowles, for her patience with this book — I hope the final result was worth the wait. I would also like to thank the many users of **CML**, who found those pesky bugs and used the language in ways that I never envisioned.

John H. Reppy
Murray Hill, NJ

[2]While a graduate student at Cornell, I was supported, in part, by the NSF and ONR under NSF grant CCR-85-14862, and by the NSF under NSF grant CCR-89-18233.

Legend



Running process
Blocked process
Spawn a process
Process termination
Synchronous communication
Sender blocks
Receiver blocks
Asynchronous communication
No blocking
Receiver blocks
Multicast communication

Contents

Preface *page* ix

Legend xv

1 Introduction 1

1.1 Concurrency as a structuring tool 2

1.2 High-level languages 6

1.3 Concurrent ML 7

2 Concepts in Concurrent Programming 11

2.1 Processes 12

2.2 Interference 13

2.3 Correctness issues in concurrent programming 14

2.4 Shared-memory languages 16

2.5 Message-passing languages 22

2.6 Parallel programming mechanisms 31

3 An Introduction to Concurrent ML 39

3.1 Sequential programming 39

3.2 Basic concurrency primitives 40

3.3 First-class synchronous operations 52

3.4 Summary 60

4 CML Programming Techniques 63

4.1 Process networks 63

4.2 Client-server programming 72

5 Synchronization and Communication Mechanisms 85

5.1 Other base-event constructors 85

5.2 External synchronous events 87
5.3 Synchronizing shared-memory 91
5.4 Buffered channels 97
5.5 Multicast channels 99
5.6 Meta-programming RPC protocols 105

6 The Rationale for CML 117
6.1 Basic design choices 117
6.2 First-class synchronous operations 120
6.3 Extending PML events 124
6.4 The expressiveness of CML 127
6.5 Discussion 129

7 A Software Build System 131
7.1 The problem 131
7.2 The design 133
7.3 Building the dependency graph 134
7.4 Creating the graph nodes 137
7.5 Parsing makefiles 139
7.6 Putting it all together 140

8 A Concurrent Window System 145
8.1 Overview 146
8.2 Geometry 146
8.3 The display system 147
8.4 The Toy Window System architecture 152
8.5 Some simple components 156
8.6 The implementation of a window manager 163
8.7 A sample application 173

9 A CML Implementation of Linda 183
9.1 CML-Linda 184
9.2 An implementation overview 187
9.3 The protocols 190
9.4 The major components 193
9.5 The network layer 194
9.6 The server layer 204
9.7 The client layer 215

10 Implementing Concurrency in SML/NJ 221
10.1 First-class continuations 221
10.2 Coroutines 223
10.3 Shared-memory concurrency 225
10.4 Simple message passing 231
10.5 First-class synchronous operations 236
10.6 Scheduling issues 244
Appendix A **A CML Reference** 249
Appendix B **The Semantics of CML** 275
Bibliography 293
Index 301

1
Introduction

Concurrent programming is the task of writing programs consisting of multiple independent threads of control, called processes. Conceptually, we view these processes as executing in parallel, but in practice their execution may be interleaved on a single processor. For this reason, we distinguish between concurrency in a programming language, and parallelism in hardware. We say that operations in a program are *concurrent* if they can be executed in parallel, and we say that operations in hardware are *parallel* if they overlap in time.

Operating systems, where there is a need to allow useful computation to be done in parallel with relatively slow input/output (I/O) operations, provide one of the earliest examples of concurrency. For example, during its execution, a program P might write a line of text to a printer by calling the operating system. Since this operation takes a relatively long time, the operating system initiates it, suspends P, and starts running another program Q. Eventually, the output operation completes and an *interrupt* is received by the operating system, at which point it can resume executing P. In addition to introducing parallelism and hiding latency, as in the case of slow I/O devices, there are other important uses of concurrency in operating systems. Using interrupts from a hardware interval timer, the operating system can multiplex the processor among a collection of user programs, which is called *time-sharing*. Most time-sharing operating systems allow user programs to interact, which provides a form of user-level concurrency. On multiprocessors, the operating systems are, by necessity, concurrent programs; furthermore, application programs may use concurrent programming to exploit the parallelism provided by the hardware.

One important motivation for concurrent programming is that processes are a useful abstraction mechanism: they provide encapsulation of state and control with well-defined interfaces. Unfortunately, if the mechanisms for concurrent programming are too expensive, then programmers will break the natural abstraction boundaries in order to ensure acceptable performance. The concurrency provided by operating system processes is the most widespread mechanism for concurrent programming. But, there are several disadvantages with using system-level processes for concurrent programming: they are expensive to create and require substantial memory resources, and the mechanisms for interprocess communication are cumbersome and expensive.[1] As a result, even when faced with a naturally concurrent task, application programmers often choose complex sequential solutions to avoid the high costs of system-level processes.

Concurrent programming languages, on the other hand, provide notational support for concurrent programming, and generally provide lighter-weight concurrency, often inside a single system-level process. Thus, just as efficient subroutine linkages make procedural abstraction more acceptable, efficient implementations of concurrent programming languages make process abstraction more acceptable.

1.1 Concurrency as a structuring tool

This book focuses on the use of concurrent programming for applications with naturally concurrent structure. These applications share the property that flexibility in the scheduling of computation is required. Whereas sequential languages force a total order on computation, concurrent languages permit a partial order, which provides the needed flexibility.

For example, consider the **xrn** program, which is a popular UNIX program that provides a graphical user interface for reading network news. It is both an example of an interactive application and of a distributed-systems application, since it maintains a connection to a remote news server. This program has a rather annoying "feature" that is a result of its being programmed in a sequential language.[2] If **xrn** loses its connection to the remote news server (because the server goes down, or the connection times out), it displays a message window (or "*dialog box*") on the screen to inform the user of the lost connection. Unfortunately, after putting up the window, but before writing the message, **xrn** attempts to reestablish the connection, which causes it to hang until the server comes back on line. Thus, you have the phenomenon of a blank message window appearing on the user's screen, followed by a long pause, followed by the simultaneous display of two messages: the first saying that the connection has been lost, and a second saying that the connection has been restored. Besides being an example of poor interface design, this illustrates the kind of sequential orderings that concurrent programming easily avoids.

[1]It should be noted that most recent operating systems provide support for multiple threads of control inside a single protection domain.

[2]This anecdote refers to version 6.17 of **xrn**.

1.1.1 Interactive systems

Interactive systems, such as graphical user interfaces and window managers, are the primary motivation of much of the work described in this book, and the author believes that they are one of the most important application areas for concurrent programming. Interactive systems are typically programmed in sequential languages, which results in awkward program structures, since these programs are naturally concurrent. They must deal with multiple asynchronous input streams and support multiple contexts, while maintaining responsiveness. In the following discussion, we present a number of scenarios that demonstrate the naturally concurrent structure of interactive software.

User interaction

Handling user input is the most complex aspect of an interactive program. An application may be sensitive to multiple input devices, such as a mouse and keyboard, and may multiplex these among multiple input contexts (*e.g.*, different windows). Managing this many-to-many mapping is usually the province of *User Interface Management System* (UIMS) toolkits. Since most UIMS toolkits are implemented in sequential languages, they must resort to various techniques to emulate the necessary concurrency. Typically, these toolkits use an *event-loop* that monitors the stream of input events and maps the events to *call-back* functions (or *event handlers*) provided by the application programmer. In effect, this structure is a poor-man's concurrency: the event-handlers are coroutines, and the event-loop is the scheduler.

The call-back approach to managing user input leads to an unnatural program structure, known as the "*inverted program structure*," where the application program hands over control to the library's event-loop. While event-driven code is sometimes appropriate for an application, this choice should be up to the application programmer, and not be dictated by the library.

Multiple services

Interactive applications often provide multiple services; for example, a spreadsheet might provide an editor for composing macros, and a window for viewing graphical displays of the data, in addition to the actual spreadsheet. Each service is largely independent, having its own internal state and control-flow, so it is natural to view them as independent processes.

An additional benefit of using process abstraction to structure such services is that it makes replication of services fairly easy. This is because processes are reentrant by their very nature (*i.e.*, they typically encapsulate their own state). In our spreadsheet example, supporting multiple data sets or graphical views should be as easy as spawning an additional process.

This is a situation where an "object-oriented" language might also claim benefits, since