2
The Essence of Concurrent ML

Prakash Panangaden and John Reppy

ABSTRACT Concurrent ML (CML) is a programming language that inte-
grates high-level abstraction mechanisms with concurrency primitives. Like
other similar, recent languages, CML embodies the idea that concurrent
programming can be done in a modern high-level language like ML. The
fundamental new ingredient is the notion of a (polymorphic) event. This is
a piece of code packaged as a new abstraction. Combinators are provided
to build new events from old ones. The key subtlety is the interaction of
external choice with abstraction. Ordinary functional abstraction conceals
the communication actions on which choices need to be made. The event
mechanism allows one to build abstractions that expose the communica-
tion. The language CML features mobility, polymorphism, and ordinary
lambda-abstraction as well as events. In this article we discuss the basic
features and explain how CML can be used to implement new concurrent
abstractions.

2.1 Introduction

At the heart of any formalism for programming or reasoning about con-
current systems is the notion of interaction between autonomous entities.
When formalisms for concurrency differ in an essential way, the interac-
tion mechanism is invariably the central point of difference. In concurrent
programming, interaction manifests itself through communication and syn-
chronization. Because of the presence of several processes executing inde-
pendently, the possible execution scenarios are subject to the complexity
of combinatorial explosion, and one needs methods for coping with the
resulting complexity.

Abstraction is a key tool for managing complexity. In the development
of software, a fundamental advance in programming languages was making
procedural abstraction and data abstraction available to the user. This has
proved to be enormously useful in practice; one is hard-pressed to imagine a
serious language without procedures. The theoretical tool needed to analyze
procedural abstraction, namely the \( \lambda \)-calculus, had already been developed
by logicians. There followed almost thirty years of fruitful (if not always
amicable) interactions between theoreticians and practitioners. One of the
high points of this development was the wonderful paper by John Reynolds,
called "The essence of Algol," which spelled out the key ideas in Algol [153].

What is the right notion of abstraction for concurrent programming? This is a question that has been asked surprisingly infrequently. Current programming practice largely adheres to the idea that "systems" programming requires working in a low-level language. Such languages provide abstractions of the underlying hardware but very little in the way of support for programmer-defined abstractions. Theoretical research in concurrent programming has been primarily focused on the algebraic properties of process calculi. When abstraction mechanisms are considered at all, they are usually attempts to combine lambda-abstraction with familiar process calculi. But as we shall argue below, this is not always the appropriate mechanism for concurrent programs. On the other hand, concurrent programs are notoriously complex, even when not particularly large; thus the need for an abstraction mechanism is particularly acute.

The languages Concurrent ML (CML) [150] and its precursor PML [149] provide a unique abstraction mechanism for concurrent programming. The key idea is to separate the operation of synchronization from the mechanism for describing synchronization and communication protocols. A new type of first-class values is introduced to represent the descriptions of synchronization and communication, and a small collection of combinators is provided for creating more complex protocols from primitive ones. This mechanism allows application programmers and library providers to design and implement new synchronization and communication abstractions that have the same status as the built-in primitives.

CML is not another process calculus with a new set of combinators; the new feature of the language is the new abstraction mechanism and the way that it is smoothly integrated with the higher-order features and polymorphic type discipline of Standard ML. Indeed, the primitives are fairly conventional: rendezvous-style communication (with value passing) as in CCS or CSP, selective choice between events, concurrent spawning of processes and the standard constructs of a sequential language. Since the language is an extension of Standard ML, the usual notion of higher-order entities is available (and used!). We emphasize, however, that CML is not just "a higher-order functional language with concurrency features." The new abstraction mechanism that is present in CML is specifically designed for concurrent programming and cannot be built out of λ-abstraction.

For the benefit of the theoretically-oriented reader, the most important point to be made about CML is that the language was designed to be used in large-scale concurrent programming. Thus, the key design decisions were not made in order to produce a pleasing algebra, but nevertheless, certain mathematically pleasant properties naturally emerged. From a foundational point of view, the choice of primitives in CML is guided by the desire to build other concurrency abstractions on top of the given primitives. In a forthcoming book on CML, it is shown how a wide variety of concurrency primitives can be expressed in CML [148].