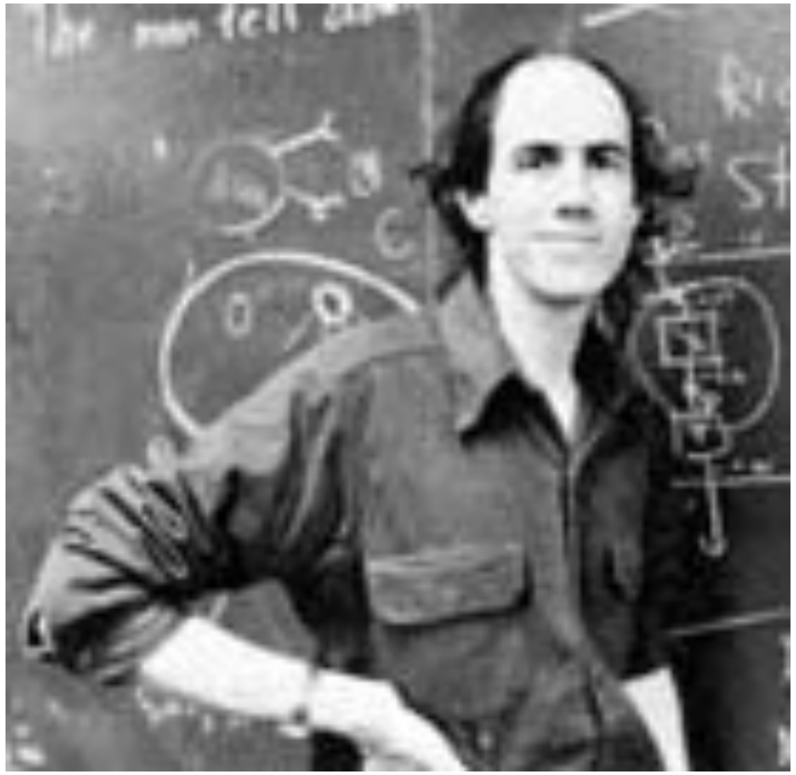
Erlang and Go concurrency

MARK ALLEN

BASHO

@BYTEMEORG

MRALLEN1@YAHOO.COM



A Universal Modular ACTOR Formalism for Artificial Intelligence

Carl Hewitt
Peter Bishop
Richard Steiger

Abstract

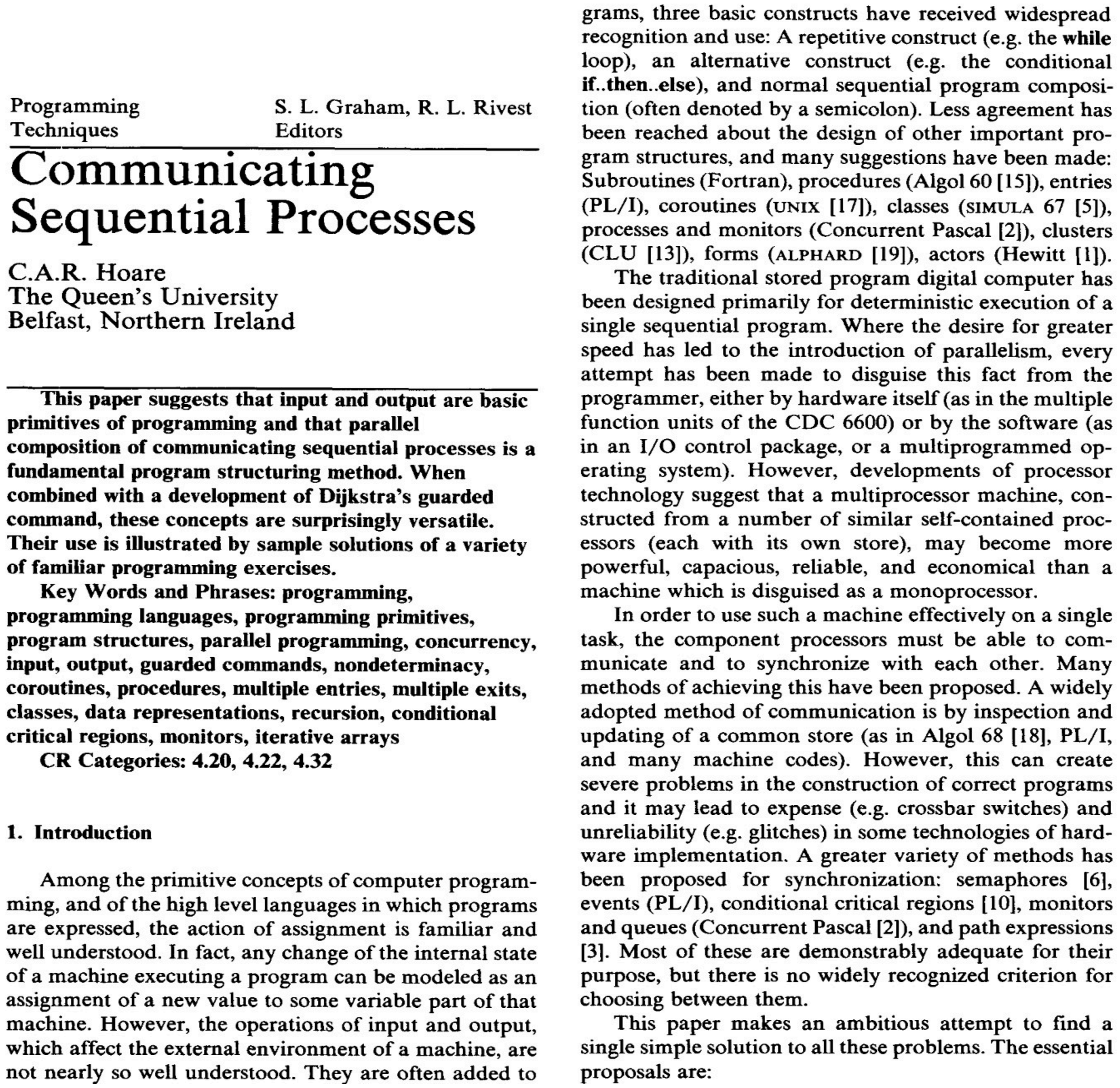
This paper proposes a modular ACTOR architecture and definitional method for artificial intelligence that is conceptually based on a single kind of object: actors [or, if you will, virtual processors, activation frames, or streams]. The formalism makes no presuppositions about the representation of primitive data structures and control structures. Such structures can be programmed, micro-coded, or hard wired 1n a uniform modular fashion. In fact it is impossible to determine whether a given object is "really" represented as a list, a vector, a hash table, a function, or a process. The architecture will efficiently run the coming generation of PLANNER-like artificial intelligence languages including those requiring a high degree of parallelism. The efficiency is gained without loss of programming generality because it only makes certain actors more efficient; it does not change their behavioral characteristics. The architecture is general with respect to control structure and does not have or need goto, interrupt, *or* semaphore primitives. The formalism achieves the goals that the disallowed constructs are intended to achieve by other more structured methods.

PLANNER Progress

"Programs should not only work,
but they should appear to work as well."
PDP-1X Dogma

The PLANNER project is continuing research in natural and effective means for embedding knowledge in procedures. In the course of this work we have succeeded in unifying the formalism around one fundamental concept: the ACTOR. Intuitively, an ACTOR is an active agent which plays a role on cue according to a script" we' use the ACTOR metaphor to emphasize the inseparability of control and data flow in our model. Data structures, functions, semaphores, monitors, ports, descriptions, Quillian nets, logical formulae, numbers, identifiers, demons, processes, contexts, and data bases can all be shown to be special cases of actors. All of the above are objects with certain useful modes of behavior. Our formalism shows how all of the modes of behavior can be defined in terms of one kind of behavior: sending messages to actors. An actor is always invoked uniformly in exactly the same way regardless of whether 1t behaves as a recursive function, data structure, or process.

"It is vain to multiply Entities beyond need."
William of Occam
"Monotheism is the Answer."



Programming Techniques

S. L. Graham, R. L. Rivest Editors

Communicating Sequential Processes

C.A.R. Hoare
The Queen's University
Belfast, Northern Ireland

This paper suggests that input and output are basic primitives of programming and that parallel composition of communicating sequential processes is a fundamental program structuring method. When combined with a development of Dijkstra's guarded command, these concepts are surprisingly versatile. Their use is illustrated by sample solutions of a variety of familiar programming exercises.

Key Words and Phrases: programming, programming languages, programming primitives, program structures, parallel programming, concurrency, input, output, guarded commands, nondeterminacy, coroutines, procedures, multiple entries, multiple exits, classes, data representations, recursion, conditional critical regions, monitors, iterative arrays

CR Categories: 4.20, 4.22, 4.32

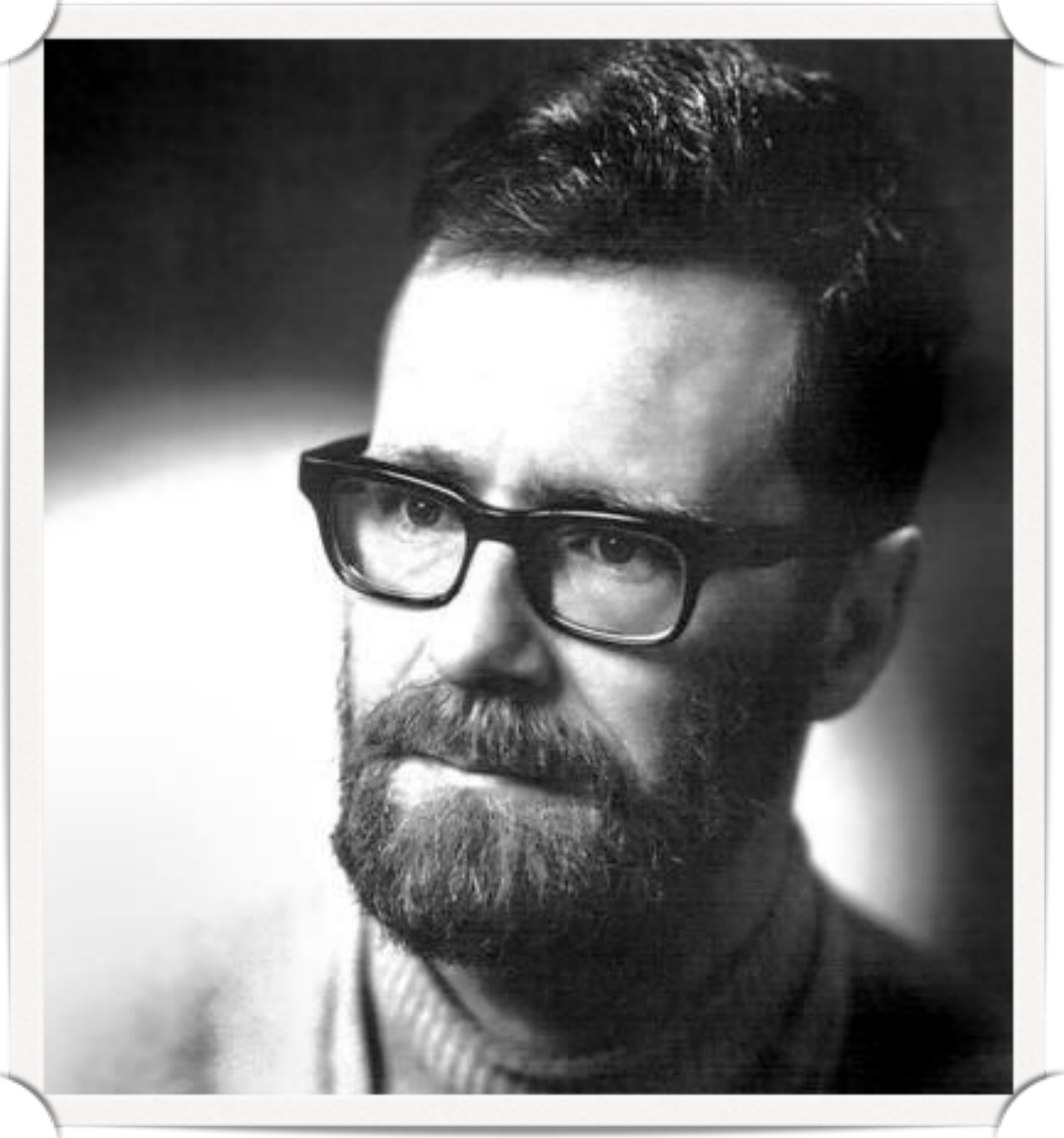
1. Introduction

Among the primitive concepts of computer programming, and of the high level languages in which programs are expressed, the action of assignment is familiar and well understood. In fact, any change of the internal state of a machine executing a program can be modeled as an assignment of a new value to some variable part of that machine. However, the operations of input and output, which affect the external environment of a machine, are not nearly so well understood. They are often added to grams, three basic constructs have received widespread recognition and use: A repetitive construct (e.g. the **while** loop), an alternative construct (e.g. the conditional **if..then..else**), and normal sequential program composition (often denoted by a semicolon). Less agreement has been reached about the design of other important program structures, and many suggestions have been made: Subroutines (Fortran), procedures (Algol 60 [15]), entries (PL/I), coroutines (UNIX [17]), classes (SIMULA 67 [5]), processes and monitors (Concurrent Pascal [2]), clusters (CLU [13]), forms (ALPHARD [19]), actors (Hewitt [1]).

The traditional stored program digital computer has been designed primarily for deterministic execution of a single sequential program. Where the desire for greater speed has led to the introduction of parallelism, every attempt has been made to disguise this fact from the programmer, either by hardware itself (as in the multiple function units of the CDC 6600) or by the software (as in an I/O control package, or a multiprogrammed operating system). However, developments of processor technology suggest that a multiprocessor machine, constructed from a number of similar self-contained processors (each with its own store), may become more powerful, capacious, reliable, and economical than a machine which is disguised as a monoprocessor.

In order to use such a machine effectively on a single task, the component processors must be able to communicate and to synchronize with each other. Many methods of achieving this have been proposed. A widely adopted method of communication is by inspection and updating of a common store (as in Algol 68 [18], PL/I, and many machine codes). However, this can create severe problems in the construction of correct programs and it may lead to expense (e.g. crossbar switches) and unreliability (e.g. glitches) in some technologies of hardware implementation. A greater variety of methods has been proposed for synchronization: semaphores [6], events (PL/I), conditional critical regions [10], monitors and queues (Concurrent Pascal [2]), and path expressions [3]. Most of these are demonstrably adequate for their purpose, but there is no widely recognized criterion for choosing between them.

This paper makes an ambitious attempt to find a single simple solution to all these problems. The essential proposals are:



Erlang concurrency

- Concurrent entities are called “processes”
- Processes have a unique node specific identifier called a “process ID”
- This is **not** the same process ID as a Unix “process” – only within the context of the Erlang run time system
- Messages are not typed (Erlang is not a strongly typed language)
- Erlang variables are immutable
- Processes share **no** memory or data with other processes
- Erlang approach to error handling is to crash on unexpected messages
- This is OK because of Erlang’s strong support for “supervisors”

Go concurrency

- Concurrent entities are known as “goroutines”
- Messages (events) are sent along typed channels
- Processes share memory
- Mutable variables

Resources

- https://github.com/mrallen1/erlang-n-go
- https://godoc.org/github.com/thomas11/csp
- http://www.informit.com/articles/printerfriendly/1768317
- http://www.thedotpost.com/2015/11/matt-aimonetti-applied-concurrency-in-go
- https://medium.com/@matryer/very-basic-concurrency-for-beginners-in-go-663e63c6ba07
- https://www.youtube.com/watch?v=sla-t0ZNlMU (Design a Real Time Game Engine in Erlang)
- https://github.com/mrallen1/parque
- https://www.youtube.com/watch?v=4STRzIrBVG8 (Early History of Distributed Systems)

Thank you!

Questions?