Actors - The Stage is Set

John R. Pugh

SCS-TR-25
June 1983

School of Computer Science
Carleton University
Ottawa K1S 5B6
Canada

This research was supported by the Natural Sciences and Engineering Research Council of Canada.

Actors - The Stage is Set

John R. Pugh

Abstract:

Proposed new software systems are always larger and more complex than their predecessors. At the same time, the programmers and designers of these systems are put under pressure to increase their productivity, keep escalating software costs in check, and produce systems which are more reliable and easy to maintain. Currently, most of this software is developed using high-level languages based on the traditional procedure-data model of computation. This paper describes an alternative which may prove to be a more appropriate model for the design of large, complex software systems. It is based on the view that software systems should be constructed as communities of active experts or "Actors", each of whom has (1) it's own knowledge base, (2) the ability to perform certain prescribed tasks and (3) is able to communicate with other Actors by message-passing. Until recently, systems based on this model have not generally been available, either tucked away in the research laboratories of major computer manufacturers, or the journals of the Artificial Intelligence community. However, as this paper describes, the model is now being used as the basis for commercial applications in areas as diverse as simulation, computer animation, and office automation.

1 Introduction

The software systems being developed by Artificial Intelligence researchers are no different in many respects from the systems being developed by the business and scientific communities. They are large, intricate systems often very difficult to implement if they must also be understandable, reliable and maintainable. The AI community has been developing their own ideas to deal with the construction of such systems; ideas whose application is not restricted to the AI domain. This paper introduces the concept of Actors, discusses their properties, and describes how Actor systems and languages are being used in application areas outside AI.

Most of today's commonly used high level programming languages are based on the classical procedure-data model of computation. They draw a distinction between the data or information being manipulated and the procedures that actually perform the manipulation. Some modern languages, Ada for example, have changed this emphasis somewhat by providing increased language support for data abstraction. This move is a recognition of the importance of the concepts of locality and modularity to the design of large software systems. An abstract data type can be thought of as a program module which extends the predefined data types provided as standard in the language. It consists therefore of a data structure together with the operations that can be applied to members of the new type bound together in a single program module. Users of the type need only be provided with a specification of the values that members of the new type may take and of the effect of each of the operations. The internal details of the data type module, data structure, details of the implementation of each operation, and local data and procedures, are hidden from the user. This supports the generally accepted idea of information hiding; presenting users with an external functional specification of a module while the internal specification remains hidden and inaccessible. This allows the internal details of a module to be modified without impact on any of it's users.

Actor or object-oriented systems take the view that a single entity, an Actor or object, should be used to represent both data and procedures. They utilise and extend the explicit facilities for creating classes and instances which were first introduced by the programming language Simula [5]. However, whereas the object-oriented facilities in Simula can be thought of as being extensions to a predominantly traditional Algol-like procedure-data language, Actor systems adopt the far more radical view that programs should consist solely of communicating actors or objects.

The first truly actor or object-oriented systems evolved from two research efforts; the development of the Smalltalk [17] family of languages by the Learning Research Group at Xerox PARC, and the development of the actor model of computation by Hewitt's research

group at the AI laboratory at MIT [6,7].

Smalltalk is much more than a programming language; it provides a complete programming environment for the design and implementation of software systems. The system is designed for use with a powerful personal computer, incorporating a high resolution bit-mapped graphics display together with a mouse and keyboard. The latest iteration of the language, Smalltalk-80 [17], has recently been made available on the Xerox Dolphin and Dorado computers.

Languages which have emerged from the AI world include Act 1 [10], an experimental language for constructing actor systems and studying the actor model of computation, Director [8], an actor-based animation language, Flavors [3], a sublanguage of Lisp Machine Lisp, ROSS [13], a simulation langauge, and LOOPS [1], which adds facilities for data, object and rule-oriented programming to Interlisp. AI researchers have seen the actor methodology of organising programs as a society of communicating problem-solving experts as a promising means of managing the complexity of large AI systems. Most of the AI actor systems are implemented as extensions to one of the many dialects of Lisp.

Smalltalk and the various AI systems share many common ideas and principles. However, there are differences in the manner in which these are realised; some systems, particularly the AI variety, introduce concepts unique to themselves. Rather than concentrate on these differences, the significant characteristics of Actor systems and their advantages will be summarised. First, a note on terminology. The term "Actor", first coined by Hewitt, is generally used in the AI community, whereas "Object" is used in Smalltalk. As will be seen in a moment, the term Actor better typifies the activity and dynamism associated with the concept, whereas Object is suggestive of passivity and an inanimate nature.

2 Characteristics of Actor systems

The actor model of computation is attractively intuitive; the interactions between independent entities (Actors) in a program can very often be equated to the process of interaction between humans. Each actor in a system can be thought of as playing out an active (acting) role not unlike the role humans play in real-life systems. The actor style of programming, sometimes called anthropomorphic programming, is one which comes very naturally to programmers. End-users of Actor-based systems enjoy the benefits of being able to think about actors in a computerised system in the same way as the corresponding physical actors in a real-life system. For example, in an electronic office environment, the office principal might read his mail by asking the mail administrator actor "Do you have any new mail for me?", or check his appointments by asking the diary administrator actor,

"What appointments do I have today?". The office principal himself can be thought of as just another actor.

What is an actor? An actor is a small processor which is defined solely by it's behavior. It's behavior is characterised by its response to receiving a message. Communication between actors is achieved through the single, uniform metaphor of message passing. When a message is received by an actor, the actor determines whether it recognises the message as one for which it has a programmed response. If so, the script, method, or procedure associated with the message is evaluated and a response, if necessary, is relayed back to the sender of the original request (or another actor). An actor has a local body of knowledge which it can use to respond to a message. If appropriate, the actor can provide operations which would allow other actors to interrogate and (possibly) update this knowledge. The actor "owns" it's own knowledge; it decides if and when other actors may access it, or even know of it's existence. A distinctive feature of actor systems is that there is no central database facility. Rather, knowledge is embedded in individual actors and thus distributed throughout a system.

Looked at in isolation, an actor consists of a local knowledge base, and a set of behaviors. A behavior is characterised by a message and a script or method defining what the actor's response to that message will be. In order to respond to a message, an actor will often need to seek assistance from other actors it is acquainted with. Thus the scripts for an actor involve sending messages to other actors and in this way messages propagate throughout an actor system. Note that, since it is the actor receiving a message that determines the response, the same message may be sent to different actors and result in many different responses. The type of messages which may be sent to actors varies greatly from system to system. The Smalltalk family uses message-sending expressions which must conform to fairly rigid syntactic rules and in many ways resemble traditional procedure calls. The message specifies a receiver, a message selector for the receiver to identify, and any arguments necessary for the processing of the message. The A.I. systems, on the other hand, often allow quite complex message structures to be used. These systems make use of sophisticated pattern matchers to parse incoming messages and have led to the development of highly readable English-like user interfaces to actor systems.

Continuing the analogy with interacting human systems, just as humans or objects often share similar characteristics and abilities, all actor systems provide mechanisms which allow actors to share common knowledge and behaviors. This allows, for instance, an actor to be described as a specialisation or extension of another actor, perhaps being exactly similar to an existing actor except that the specialised actor has additional knowledge or behaviors. Knowledge sharing avoids the duplication of scripts for shared behaviors in every actor and makes the modification of such scripts more manageable by centralising them in a single actor.

There is no uniform mechanism for the implementation of knowledge sharing within actor systems. A common approach is to use the notion of an actor inheriting the knowledge and behavior of another. The simplest form of inheritance arranges actors in a hierarchy or tree structure. Actors automatically inherit the knowledge and behavior of their parent in the tree. In such an environment, an actor receiving a message which it itself cannot respond to, will pass the message on to it's parent which may be able to respond to the original message or who in turn may pass the message on to it's parent. The actor at the root of the tree is the most general and contains behaviors applicable to all actors. Individual actors may override the automatic inheritance mechanism by defining their own individual behaviors or knowledge which will supercede those contained within the parent. In Smalltalk, inheritance is implemented using a subclassing mechanism. Instances of a class inherit the attributes of that class. If that class is a subclass of another then the instance may inherit attributes of the superclass if needed. The actor languages often base the inheritance mechanism on ancestral information. An actor inherits the capabilities of the actor which created it. The simple inheritance mechanism based on ancestral information has been found to be too restrictive for efficiently describing many problem situations. Actors are often not simply specialisations or extensions of a single actor; rather they embody facets or characteristics from a number of actors. Thus, mechanisms which support multiple inheritance, the ability to inherit knowledge from more than one actor, have been incorporated into many actor systems [2,4,10,12].

Returning to the analogy with real-life systems one final time, our discussion has so far neglected one very important fact; humans in real-life systems operate concurrently. Primitives for dealing with the problems of concurrency have traditionally been grafted on to procedure-data languages (e.g. Ada, Modula, Concurrent Pascal). One of the main motivations for much of the work on actor systems has been the belief that they are far better placed to exploit and control the parallelism which must be harnessed by large AI systems of the future. For example, the problems of access to shared resources are more easily controlled in an actor world where knowledge is centralised in resource management actors, and where concurrently executing processing units (actors) are largely independent and communicate solely via message-passing.

3 Applications

To illustrate the broad range of application areas for which the actor paradigm is appropriate and to provide references to existing actor-based systems, we will briefly describe their use in three application areas: simulation, animation, and office automation.

Since actor systems view programming as a simulation of the the real world, simulation

is an obvious application area for actors. A research program in knowledge-based simulation at the Rand Corporation has led to the development of the actor-oriented language ROSS [13]. ROSS has been successfully used as the implementation language for SWIRL [9], an air battle simulator for use by military strategists. Objects in the air battle domain, for example, AWACS systems, fighter planes, enemy 'penetrators', command centres and fighter bases, are represented as ROSS actors in the simulation system. Actors are either generic actors or instance actors. The generic Fighter class actor, for example, contains knowledge and scripts common to all Fighters, while individual Fighter actors contain instance specific information. Generic knowledge and scripts are accessed through a multiple inheritance mechanism. Messages to actors are pattern matched, allowing a readable and flexible English-like command structure appropriate for military personnel who are concerned only with the air battle domain.

The actor model of computuation has been used extensively in computer animation systems such as ASAS [14], DIRECTOR [8], and CINEMIRA [15]. These systems use actors to represent theatrical performers and animated objects and allow the movement of these actors to be programmed independently in such a way that they appear to move concurrently from one movie frame to the next. ASAS (Actor/Scriptor Animation System), initially developed at MIT, was reponsible for the stunning special effects sequences in the recent movie TRON. ASAS is a general-purpose programming language, built on top of LISP, which has been extended with primitives for geometric objects and operators and parallel control structures. When an animated sequence is produced frame by frame, ASAS allows actors to enter and exit 'on cue', act in parallel, and synchronise with each other through message-passing.

In the area of office automation, the implementation of the revolutionary graphical user interface to the Xerox Star [12] professional workstation is based on the principles of object-oriented programming, subclassing and multiple inheritance. The workstation software comprises over a quarter of a million lines of code. Before a single line of code was written, it was decided that users would best relate to the final system if it were designed around the metaphor of the physical office. This was achieved through a user interface which, using a high-resolution bit-mapped display, presents users with a virtual desktop on which iconic representations of physical office objects such as filing cabinets, appointment books, and mail trays can be manipulated. The selection of an object-oriented implementation was a natural one. It made possible the situation where the system design metaphor of communicating physical office objects was also the model on which the implementation was based.

4 Summary

We believe that, when actor and object-oriented languages become more generally available, the actor paradigm will have a profound influence on programming practice.

The actor model may also prove to be more appropriate in exploiting the parallelism presented by advanced VLSI computer architectures. We have discussed the notion that actors can be thought of as independent processing units operating concurrently with their own 'instruction sets', 'memory', and the ability to communicate with other actors by message-passing. As memory and processor costs fall, it is possible to seriously envisage machines whose computational power will be derived from interconnected processors where each processor has it's own memory. Using this kind of architecture, actors may be assigned to different processors and work on the solution to a problem in parallel. At least one such system [10] is currently under construction. Of course, the problems associated with this type of distributed computation are far from clearly understood at the present time.

Finally, we would like to acknowledge the work of those researchers without whom this paper could not have been written and express our thanks to Professor Wilf Lalonde for his valuable comments on early drafts of this paper.

5 References

1. Bobrow, D.G., and Stefik, M.J. *The LOOPS Manual (Preliminary Version)* Knowledge-based VLSI Design Group Technical Report, KB-VLSI-81-13, Stanford University, August 1984.
2. Borning, A. and Ingalls, D.H. *Multiple Inheritance in Smalltalk-80* Proceedings of the AAAI Conference. Pittsburgh. Aug 1982.
3. Cannon, H.I. *Flavors.* Technical Report MIT Artificial Intelligence Lab. 1980.
4. Curry, G., Baer, L., Lipkie, D., and Lee, B. *Traits: An approach to Multiple-Inheritance Subclassing.* Proceedings ACM SIGOA Conference on Office Information Systems, published as ACM SIGOA Newsletter Vol 3, Nos 1 and 2, 1982.
5. Dahl, O-J., and Nygaard, K. *SIMULA - An Algol-based Simulation Language.* CACM, Vol 9, No 9, 1966, pp 671-678.
6. Hewitt, C. *Protection and Synchronization in Actor Systems,* ACM SIGCOMM-SIGOPS Interface Workshop on Interprocess Communication March 24-25, 1975. Santa Monica, California.
7. Hewitt, C. *Viewing Control Structures as Patterns of Passing Messages.*

A.I. Journal, Vol. 8, No. 3, June 1977, pp. 323-364.

8. Kahn, K. *DIRECTOR Guide*. MIT AI Memo 482B, Dec. 79.

9. Klahr, P., McArthur, D., and Narian, S. *SWIRL: An Object-Oriented Air Battle Simulator*. Proc. AAAI-82, Pittsburgh, August, 1982.

10. Lieberman, H. *A Preview of ACT 1*. MIT AI Laboratory Memo No 625, June 1981.

11. Lieberman, H. *Thinking about lots of things at once without getting confused - Parallelism in Act 1* MIT AI Laboratory Memo No 626, May 1981.

12. Lipkie, D.E., Evans, S.R., Newlin, J.K., Weissman, R.L. *Star Graphics: An Object-Oriented Implementation*. ACM Computer Graphics, Vol 16, No 3, July 1982.

13. McArthur, D., and Klahr, P. *The ROSS Language Manual*. N-1854-AF, The Rand Corporation, Santa Monica, 1982.

14. Reynolds, C.W. *Computer Animation with Scripts and Actors*. Proceedings of ACM SIGGRAPH Conference. July 1982.

15. Thalmann D, Magnenat-Thalmann N. *Actor and Camera Data Types in Computer Animation*. Proc. Graphics Interface 83, Edmonton, Canada.

16. Weinreb D., Moon D. *Flavors - Message-passing in the Lisp Machine*. MIT AI Memo No. 602 Nov. 1980.

17. Xerox Learning Research Group, *The SmallTalk-80 System*, BYTE, August 1981.

CARLETON UNIVERSITY
School of Computer Science

Bibliography of SCS Reports

SCS-TR-1 THE DESIGN OF CP-6 PASCAL
Jim des Rivieres and Wilf R. LaLonde, June 1982.

SCS-TR-2 SINGLE PRODUCTION ELIMINATION IN LR(1) PARSERS:A SYNTHESIS
Wilf R. LaLonde, June 1982.

SCS-TR-3 A FLEXIBLE COMPILER STRUCTURE THAT ALLOWS DYNAMIC PHASE ORDERING
Wilf R. LaLonde and Jim des Rivieres, June 1982.

SCS-TR-4 A PRACTICAL LONGEST COMMON SUBSEQUENCE ALGORITHM FOR TEXT COLLATION
Jim des Rivieres, June 1982.

SCS-TR-5 A SCHOOL BUS ROUTING AND SCHEDULING PROBLEM
Wolfgang Lindenberg, Frantisek Fiala, July 1982.

SCS-TR-6 ROUTING WITHOUT ROUTING TABLES
Nicola Santoro, Ramez Khatib, July 1982.

SCS-TR-7 CONCURRENCY CONTROL IN LARGE COMPUTER NETWORKS
Nicola Santoro, Hasan Hural, July 1982.

SCS-TR-8 ORDER STATISTICS ON DISTRIBUTED SETS
Nicola Santoro, Jeffrey B. Sidney, July 1982.

SCS-TR-9 OLIGARCHICAL CONTROL OF DISTRIBUTED PROCESSING SYSTEMS
Moshe Krieger, Nicola Santoro, August 1982.

SCS-TR-10 COMMUNICATION BANDS FOR SELECTION IN A DISTRIBUTED SET
Nicola Santoro, Jeffrey B. Sidney, September 1982.

SCS-TR-11 A SIMPLE TECHNIQUE FOR CONVERTING FROM A PASCAL SHOP TO C SHOP
Wilf R. LaLonde, John R. Pugh, November 1982.

SCS-TR-12 EFFICIENT ABSTRACT IMPLEMENTATIONS FOR RELATIONAL DATA STRUCTURES
Nicola Santoro, December 1982.

SCS-TR-13 ON THE MESSAGE COMPLEXITY OF DISTRIBUTED PROBLEMS
Nicola Santoro, December 1982.

SCS-TR-14 A COMMON BASIS FOR SIMILARITY MEASURES INVOLVING TWO STRINGS
R.L. Kashyap and B.J. Oommen, January 1983.

SCS-TR-15 SIMILARITY MEASURES FOR SETS OF STRINGS
R.L. Kashyap and B.J. Oommen, January 1983.

SCS-TR-16 THE NOISY SUBSTRING MATCHING PROBLEM
R.L. Kashyap and B.J. Oommen, January 1983.

SCS-TR-17 DISTRIBUTED ELECTION IN A CIRCLE WITHOUT A GLOBAL SENSE OF ORIENTATION
E. Korach, D. Rotem, N. Santoro, January 1983.

.../2

SCS-TR-18 A GEOMETRICAL APPROACH TO POLYGONAL DISSIMILARITY AND THE CLASSIFICATION OF CLOSED BOUNDARIES
R.L. Kashyap and B.J. Oommen, January 1983.

SCS-TR-19 SCALE PRESERVING SMOOTHING OF POLYGONS
R.L. Kashyap and B.J. Oommen, January 1983.

SCS-TR-20 NOT-QUITE-LINEAR RANDOM ACCESS MEMORIES
Jim des Rivieres, Wilf LaLonde and Mike Dixon, August 1982, Revised March 1, 1983.

SCS-TR-21 SHOUT ECHO SELECTION IN DISTRIBUTED FILES
D. Rotem, N. Santoro, J. B. Sidney, March 1983.

SCS-TR-22 DISTRIBUTED RANKING
E. Korach, D. Rotem, N. Santoro, March 1983

SCS-TR-23 A REDUCTION TECHNIQUE FOR SELECTION IN DISTRIBUTED FILES : I
N. Santoro, J. B. Sidney, April 1983.

SCS-TR-24 LEARNING AUTOMATA POSSESSING ERGODICITY OF THE MEAN : THE TWO ACTION CASE
M.A.L. Thathachar and B. J. Oommen, May 1983

SCS-TR-25 ACTORS - THE STAGE IS SET .
John R. Pugh, June 1983

SCS-TR-26 ON THE ESSENTIAL EQUIVALENCE OF TWO FAMILIES OF LEARNING AUTOMATA
M. A. L. Thathachar and B. J. Oommen, May 1983