HAL: A High-level Actor Language and Its Distributed Implementation

Chris Houck*and Gul Agha†
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
Email: {houck | agha}@cs.uiuc.edu

Abstract

We describe HAL, a high-level, actor-based language which has served as a test-bed for experimenting with powerful linguistic constructs for parallel and distributed programming. HAL is an architecture independent, concurrent object-oriented language which supports inheritance, synchronization constraints, continuation capture, synchronous and asynchronous message passing, and reflection. The HAL compiler has been used to execute actor programs on a number of shared and distributed memory machines. HAL allows powerful abstractions to be defined and reused. Furthermore, HAL supports the use of synchronization constraints at a fine-grained level and the use of software pipelining. In this paper, we describe the design of HAL, using specific examples to illustrate its features. We then discuss some of the implementation issues in the run-time system.

Keywords: Actors, Concurrency, Object-Oriented Languages, Distributed Execution.

1 Introduction

The actor model is a flexible model of concurrent computation in distributed systems [1]. Specifically, actors can be thought of as an abstract representation for multicomputer architectures. Although, the actor model is fairly low-level, it unifies object-oriented and functional programming and allows a rich set of abstractions to be built. Actors are dynamically reconfigurable and extensible and thus suitable for modeling open systems.

A number of actor languages have been implemented; these implementations have generally used a simulation of distributed execution on a single processor [17, 18, 23]. Although such simulations are a useful tool for some purposes, a number of issues such as load balancing and distributed garbage collection are harder to realistically emulate. On the other hand, a number of other concurrent languages available on distributed memory machines are generally not portable [6, 12], thus limiting the their utility.

This paper describes an architecture independent actor system called HAL. HAL programs have been run on shared memory, distributed memory and uniprocessor architectures. We focus on issues of compilation and the necessary run-time support needed to execute actor based programs. Architecture independence has been achieved by compiling HAL source code down to an existing architecture independent parallel processing environment called CHARM [8].

HAL has served as a test-bed for experimenting with new language constructs and dependability methods. This paper discusses some of the more interesting linguistic extensions that have been made to the basic actor model. We have found that these extensions greatly increase the flexibility and reusability of HAL programs. In particular, new mechanisms for *inheritance*, *reflection* and *synchronization constraints* have been added. Because each of these mechanisms needs to address problems arising from its interaction with concurrency, they differ from their sequential counterparts. We discuss the details of the design later in the paper.

This paper is organized as follows. Section 2 gives a brief overview of the actor model and a short description of the syntax of HAL. Section 3 contains a comparison with other distributed object-based pro-

*Author's current address: National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign.

†The work described in this paper has been made possible by generous support provided by a Young Investigator Award from the Office of Naval Research (ONR contract number N00014-90-J-1899), by an Incentives for Excellence Award from the Digital Equipment Corporation, and by joint support from the Defense Advanced Research Projects Agency and the National Science Foundation (NSF CCR 90-07195).

gramming systems. Linguistic extensions to the actor model are outlined in Section 4. Examples using HAL as a programming language are given in Section 5. Section 6 discusses our compiler and run-time system implementation. Conclusions and future directions are outlined in Section 7.

2 The Actor Model

Actors are self-contained, interacting, independent components of a computing system that communicate by asynchronous message passing [1]. Each actor has a *mail address* and a *behavior*. The *acquaintances* of a given actor are the actors whose mail addresses are known by the actor. The mail addresses of actors may be contained in messages, leading to a dynamic actor interconnection topology. In order to abstract over processor speeds and allow adaptive routing, preservation of message order is not guaranteed. However, messages sent are guaranteed to be received with an unbounded but finite delay. New actors may be created dynamically thus allowing continuations to be modeled. Actor creation and reconfiguration supports flexible, incremental construction of distributed systems.

State change in actors is specified by *replacement behaviors*. Each time an actor processes a communication, the actor computes the behavior it will exhibit in response to the next communication. The replacement behavior for a purely functional actor is identical to the original behavior; in general the behavior may change. The change in the behavior of an actor may represent a simple change of state variables, such as change in the balance of a bank account, or it may represent a change in the class of an actor. For example, suppose a bank account actor accepts a withdrawal request. In response, it will compute a new balance which will be used to process the next message. If the same bank account actor receives a request to become a pizza-delivery actor, it may decide to change its entire structure.

The HAL language

The actor model is a general framework of computation. In order to experiment with distributed implementations of actor programs, we have developed a high-level actor language HAL. HAL is an object-oriented language with a lisp-like syntax which is compiled to C code and executed using the CHARM programming system [15]. The full syntax of HAL can be found in [11]. HAL is loosely based on the previous actor languages Rosette [18], Acore [17] and ABCL/1[23].

In HAL, there are two primitives to modify an actor's state. A change in an actor's behavior definition is specified through the `become` command. For example, changing a Bank Account actor to a Pizzeria actor or changing the number of local variables an actor knows requires the `become` statement. The syntax of `become` is:

```
(become <class-name> <expr>* )
```

The actor's new class is `<class-name>`. The new bindings for *all* of its local variables are given by `<expr>*`. The `update` statement can be used to specify less dramatic changes in an actor's state. The expression `(update A B)` binds the variable `A` to `B` for the next message (actor state changes only go into effect for the *next* message that they process [1]).

The canonical example of actor behavior is a simulation of a bank account. A bank account may change its state as money is `deposited` or `withdrawn`. In addition, a bank account may be shared between one or more actors. This sharing of mutable objects is what prevents functional languages from effectively modeling an object such as a bank account. Figure 1 defines a `Bank-Account` actor class in the syntax of HAL. An instance of this class will accept `deposit`, `withdraw` and `get-balance` messages. A new `Bank-Account` with a balance of 1000 units is created with the expression:

```
(new Bank-Account 1000)
```

One may deposit 150 units into a `Bank-Account` bound to the variable `account` by the command:

```
(send deposit account 150)
```

The account may be shared with one's spouse with a message of type `our-account` by the command:

```
(send our-account spouse account)
```

Now, whenever either party interacts with the account, both actors may see the results.

3 Related Work

The HAL system has benefited from a large body of research in concurrent object-based programming languages. Some of the important aspects of the more relevant projects are described below.

Inheritance has been found to be an effective means of expressing classifications between objects and supporting modular reuse of code. Unfortunately, many

```
(define-Actor Bank-Account
  (slots balance)
  (method (deposit amount)
     (update balance (+ balance amount))
     (print "Deposited %d\n" amount))
  (method (withdraw amount)
     (let* [[newBal (- balance amount)]]
        (update balance newBal)
        (print "Withdrew %d\n" amount)
        (print "Balance  %d\n" newBal)))
  (method (get-balance)
     (print "Balance %d\n" balance))
```

Figure 1: A `Bank-Account` actor class

concurrent programming languages, (e.g., CHARM [15], Acore [17], Cantor[6], Emerald[12] and ABCL/1 [23]) do not incorporate inheritance. In our view, it is important to study the use of inheritance in concurrent systems. In particular, synchronization constraints (described below) can interfere with inheritance in concurrent systems.

It is often desirable to place some form of *synchronization constraint* on objects to maintain their internal consistency. In Rosette[18], these constraints take the form of enabled-sets which are specified inline. This mixing of code and enabledness conditions greatly reduces the reusability of inherited code. ABCL/1 also places synchronization issues within the scope of method definitions. In POOL[5], the *body* of an object is responsible for maintaining object consistency. However, object bodies are not inheritable; therefore, synchronization constraints must be repeatedly specified.

Another important aspect of a language's programmability is the types of message passing styles supported. Rosette, ABCL/1 and Acore are the only languages that provide support for both synchronous and asynchronous message passing styles. CHARM and Cantor allow only asynchronous message passing while, in Emerald, all invocations are synchronous. The POOL languages vary in their support for message passing styles. It appears as though some dialects (POOL2) allow only asynchronous message passing [4], while others (POOL-I) allow only synchronous messages [5].

Perhaps the most surprising aspect of these concurrent languages is that many of them are not portable. For example, ACBL/1, Rosette and Acore are run on uniprocessor virtual machines (though ABCL/1 is currently being ported to the EM-4 platform). Cantor, POOL and Emerald can be run in a limited number of distributed environments. CHARM is one of the few languages that is reasonably architecture independent. This fact was the primary motivation for basing the HAL run-time system on CHARM.

4 Language Features

The basic actor model [1] is a fairly low-level model, without predefined abstraction mechanisms and linguistic features necessary for non-trivial programming. We have therefore introduced a number of syntactic constructions to make programming easier. High level descriptions of these features will be discussed in the rest of this section. All the additional functionality we support could be simulated in terms of the basic actor primitives, but this would exact a great cost in terms of readability and modularity.

A conscious effort has been made to make HAL more of a class-based language than Rosette [18]. As a result, HAL satisfies all of Wegner's qualifications to be classified as object-oriented [22]. For example, the `define-Actor` construct defines a class of actors; every actor belongs to a class. However, `new` (instance creation) and `suicide` (instance destruction) are the only class methods. In addition, class variables go against the actor paradigm of avoiding shared state between actors; thus they are not supported.

4.1 Constraints

In order to maintain internal consistency, it is often the case that an actor is unable to process a message as soon as the message arrives. Going back to the bank account example, this may occur if a `withdraw` message arrives when the account balance is zero. Some form of *synchronization constraint* must be placed on the bank account actor to specify when `withdraw` messages are "serviceable" [10]. The following syntax is used to specify acceptance constraints as a function of the actor's state and the contents of the message:

```
(constrain <msg-expr> <expr>)
```

where `<msg-expr>` specifies a message name and bindings of message values, so that they may be used in testing the constraints. The semantics of `constrain` is that when `<expr>` evaluates to `TRUE` the actor is enabled to execute `<msg-expr>`. If multiple constraints are specified for the same method all of the constraint expressions must evaluate to `TRUE` for that method to be enabled. Note that `<expr>` is a side-effect free expression which may be a function of the actor's state variables *and* values contained in the `<msg-expr>`. If `<expr>` is `FALSE`, the mail message is added to a local `Pending` queue and re-evaluated later. Currently, whenever the state of an actor changes the constraints of all of the messages on `Pending` queue are checked, providing *strong fairness* [9]. A proposed optimization would create dependency lists between the actor's state variables and the constraints to reduce the number of constraint expressions which need to be retested.

The `(constrain...)` construct may be used to order messages of a certain type. This ability is useful in many numeric computations [3] where multiple iterations of an algorithm may be active concurrently but execution ordering must be maintained. This style of constraint specification makes code more reusable as will be seen in Section 4.4. However, it only offers a *hold/deliver* decision procedure. More generally, it may be the case that an actor knows that it will *never* be able to process a message. In the next section we introduce the concept of *message forwarding*.

4.2 Message Forwarding

Message forwarding provides a mechanism to abstract control flow and delegation choices from method definitions. Message forwarding addresses the problem that an actor may receive a message which it knows it will never be able to process. It is useful in cases such as exception handling, delegation and real-time deadline checking. As an example of where such error handling would be useful, consider the canonical bank account. If a bank account actor receives a `deliver-pizza` message it must somehow be dealt with. Actor semantics say that the actor could conceivably `become` a pizza-delivery actor at some later date. Therefore, in standard implementations, the `deliver-pizza` message would be placed on the actor's `Pending` queue.

If it can be determined that the bank account will never become a pizzeria, such a behavior is unsatisfactory. From an implementation standpoint, the message, stuck on the `Pending` queue, will take up space and waste cycles as its constraint conditions are repeatedly checked. The syntax we have developed for message forwarding is:

```
(forward <msg-expr> <msg-send> <expr>)
```

As with constraints, `<expr>` is a side-effect free expression which can use values supplied by the message to determine enabledness. If the expression is `FALSE` the `<msg-send>` is executed, otherwise it is tested against any other synchronization constraints.

Automatic forwarding may be used to provide a global service while only requiring clients to know the addresses of the local representative. In this way, we have been able to build primitive forms of Concurrent Aggregates (CA) [7]. A CA is a multi-object structure: when a message arrives at an object in a CA the object forwards the message to the members of the aggregate responsible for processing that particular message. Not having a single entry point into the aggregate structure increases the throughput of the service that the CA implements. We are currently studying high-level abstractions for CA.

4.3 Message Passing Paradigms

We have implemented two message passing constructs in addition to the basic asynchronous send. The first is a message order preserving send, or *sequenced send*. The second is a remote procedure call mechanism similar to Acore's `ask` primitive [17].

Sequenced Sends. As noted above, the `(constrain...)` expression may be used to order messages of a certain type. This construct is specified in the receiver and is primarily concerned with *types* of messages. In contrast, *sequenced sends* allow the sender to impose an arrival order on a series of messages sent to the same target. With constraints, the receiver is responsible for the order of message processing; with sequenced sends, the sender is given some control over the message reception order.

Sequenced sends are implemented by tagging and reshuffling such messages at the recipient. Thus communication overhead is reduced. The prime benefit arises when the sender requires a sequence of actions to happen in a given order when the receiver may or may not care about the order.

RPC Sends. With asynchronous communication, the programmer needs to explicitly write continuations whenever a synchronous exchange is desired. By synchronous we mean that the original sender requires some form of response from the receiver. The response can be either a value based on the original message or a simple acknowledgment that the original message was processed. HAL extends the basic actor semantics by supporting synchronous communication.

In synchronous method invocations, the calling program implicitly blocks and waits for a value to be returned from the actor whose method was invoked. Because only explicit message passing is allowed in CHARM [15], we automatically *lift* synchronous messages and their continuations out of user code in a manner akin to lambda lifting [13]. A more detailed description of this problem and our implementation is contained in Section 6.3.

4.4 Inheritance

HAL allows for inheritance of both code and synchronization constraints. However, while the code of one's ancestor may be overwritten in the style of Smalltalk[16], we have adopted the view that an ancestor's synchronization constraints may *never* be over-written; they are always in effect[10]. In this sense, inheritance is viewed as a means of *specialization.* An example of how inheritance and synchronization constraints interact will be seen in Section 5.2.

4.5 Reflection

One of the fundamental aspects of open systems is that they are *extensible.* One aspect of extensibility is the ability to dynamically change the underlying system executing a program through *reflection* [20]. Full reflection, which would allow modification of every aspect of the system down to the arithmetic interpreter, is not currently supported.

The current definition of HAL allows for an actor to replace its *dispatcher* and *mailq* (the actors responsible for sending and receiving mail messages, respectively) through the process of *reification.* The dispatcher is represented as an actor through which all outgoing mail is passed. A message to an actor is received by its (reified) mailq. Once the mailq actor processes the message, it forwards the message to the original recipient. The current reflective architecture is sufficient to implement a number of significant examples – including a meta-architecture for fault-tolerance[2].

5 Examples

5.1 Software Pipelining

The actor model provides for a means of increasing program efficiency by supporting a high degree of *software pipelining.* In [3], implementations of a Cholesky Decomposition of a symmetric positive definite matrix were explored in order to demonstrate the benefits of pipelining. Using an actor based programming style can lead to programs that monotonically improve in performance. The pipelining is natural to express: instead of global synchronization, constraints are pushed to the lowest level of granularity.

```
(define-Actor Buffer
  (slots count)
  (constrain (get) (> count 0))
  (method (get)
      (update count (- count 1)))
  (method (put)
      (update count (+ count 1)))
  (method (get-count)
      (print "Size %d" count)))

(define-Actor Bounded-Buffer
  (slots max)
  (superclass Buffer)
  (constrain (put) (> max count)))

(define-Actor Get2-Buffer
  (superclass Bounded-Buffer)
  (constrain (get2) (> count 1))
  (method (get2)
      ; Remove two elements atomically
      (update count (- count 2))))
```

Figure 2: A hierarchy of `Buffer` classes.

5.2 Inheritance of Constraints

Figure 2 defines a set of buffer classes which illustrates some of the re-use that is supported by inheritance and our constraint syntax. The most basic class in this example is a `Buffer` which has methods to `get` and `put` elements to and from the buffer and a `get-count` method which prints the number of elements currently in the buffer. The simple `Buffer` class has a constraint, namely `(> count 0)`, which must hold in order for a `get` message to be processed.

The `Bounded-Buffer` class is a sub-class of `Buffer`. In the `Bounded-Buffer`, there is a constraint on the absolute size of the buffer; there can never be more than `max` elements in the buffer at any time. Since we have separated the logic of the constraints from the code for the actual methods, we only need to state the new constraints without having to redefine the `put` method.

Finally, we define the class `Get2-Buffer`, a subclass of `Bounded-Buffer`, with the ability to remove two elements from the buffer as an atomic action. This behavior is provided through the `get2` method. In contrast, defining a method such as `get2`, in Rosette, would require reimplementing both `get` and `put` as constraints are specified through *enabled-sets* and the superclass `get` and `put` operations would

```
(define-Actor Dispatcher
  (method (default)
      (print "Sent type %d\n" msg-type)
      (send msg-type msg-dst)))

(define-Actor Me
  (method (initialize)
      (dispatcher (new Dispatcher))
  (method (...)))
```

Figure 3: Using reflection for debugging information.

never enable the `get2` method [19]. In HAL, we can simply declare the new method and the relevant synchronization constraints. Thus, the definitions of both `get` and `put` can then be inherited from the class `Buffer` in both of these subclasses.

5.3 Reflection

A practical need for reflection can be found in the area of program debugging. In order to debug a program, it is often helpful to know the order and frequency of method invocations. However, for the programmer to edit the source code and place `print` statements at every method invocation is tedious and error prone. Instead, the actor model allows a programmer to *reify* the postal system (i.e. to modify the routines responsible for both message delivery and message sending), so that whenever a message is sent or delivered, the newly reified postal system prints the relevant diagnostic information. When nolonger required, the programmer can simply remove the reflection code and return the system to normal execution.

The reflective capabilities of HAL may be used to print such information (see Figure 3). An actor of class `Me` will replace its dispatcher with an actor that will print diagnostic information to the console. Our implementation of `Dispatcher` makes no refernce to its base actor; therefore, many actors could share the same meta-dispatcher providing a limited form of groupwide reflection[21].

HAL has also been used to experiment with implementing dependability protocols at the meta level thus providing a separation of design concerns [2]. One of the design motivations for HAL was to create a platform to test out theoretical ideas. In this case, it has been clearly shown that using reflection to implement fault-tolerance is possible and, from a code complexity standpoint, desirable.

6 An Actor Run-Time System

In order to make the compiler and run-time system machine independent, the CHARM system [15] is used as the compilation target. A CHARM source program only needs to be recompiled to be transferred between many existing shared memory and distributed memory architectures [8]. The reliance on CHARM implies that a number of aspects of the actor model are sacrificed, the most serious being *internal concurrency*. The actor model allows the expressions that make up a single method to be executed concurrently [1].

It is obvious that simply building an actor compiler is insufficient for distributed execution of actor programs; a run-time system must also be provided. For example, it is necessary to provide facilities for address lookup and resource management problems among others. HAL run-time support is provided by a distributed kernel which uses the following CHARM facilities: architecture independence, dynamic process creation, transmission of addresses in messages, globally valid addresses and creation-time load balancing. Actors are load balanced at their creation time, automatic migration of existing tasks is not supported by the CHARM system, however, our implementation has been based on the assumption that it may soon be available.

Another critical aspect of run-time support is *garbage collection*. The actor model abstracts away from details of memory management; the task is thus left to the run-time system. Actor languages present a unique difficulty for resource management in that it is necessary to determine both the "reachability" and "state" (active or blocked) of an actor[14]. Automatic garbage collection in HAL has been implemented but has not yet been fully tested.

6.1 Distributed Kernel

The distributed run-time kernel contains all of the code for the methods in an actor program (see below), the kernel is, additionally, responsible for bookkeeping tasks such as checking constraints and managing pending queues. Currently, the same kernel is placed on every node, an artifact of CHARM. The conceptual view of an individual processor node is presented in Figure 4. The object labeled "interpreter" is necessary for some types of reflection and does not currently exist. Dotted lines are messages sent between actors; solid lines represent kernel function calls.

Even though it is written in C, great care has been taken to make the code for the kernel as readable and modular as possible. Routines which provide different kernel facilities are disjoint. Kernel modularity is also

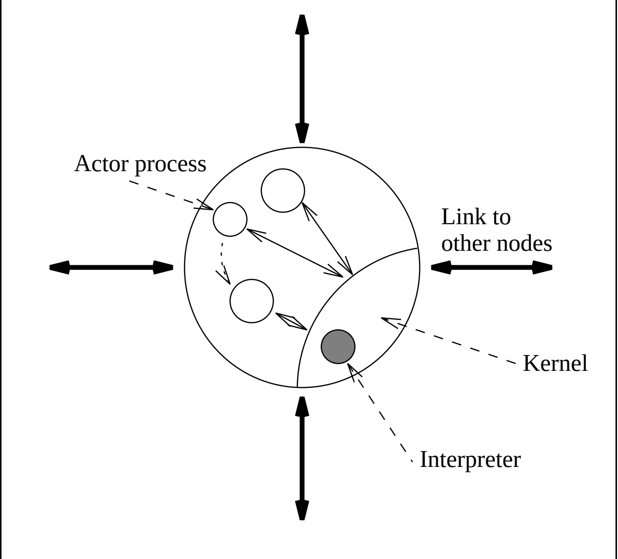

Figure 4: Conceptual view of a node.

beneficial from an efficiency standpoint: for example, code to check constraints is not generated in the kernel unless the source program contains constraint expressions. Therefore, an executing program does not call the constraint checker unless it is necessary. In fact, when a program does not contain constraints, the constraint checking routines are not even linked into the kernel, saving space.

6.2 Actor Representation

In our implementation, actors are represented as lightweight processes. The process has a single data structure representing the entire state of the actor. One part of the state record is a `Behavior` field which specifies the type of actor the process is currently representing. Method definitions are stored in the kernel. When an actor receives a message, it hands control to an interface routine in the kernel which takes the actor's state record and the new message and calls the appropriate function to execute the actor's behavior.

Since all behavior definitions are stored in the local kernel, it keeps the amount of code that has to be placed on each node at a minimum. The kernel code size is linear in the number of methods defined in a user's program. Actor processes require storage linear in the size of an actor's acquaintance list and the number of messages on the `Pending` queue. Furthermore, the actors are self-contained, that is to say that all of the state information pertaining to a particular actor is accessible from a single data structure.

An individual actor is represented as a single record containing its state; a pointer to this record is then passed to the kernel on message reception. This creates a special problem with state changing expressions (`become` and `update`). Actor semantics say that these commands only take effect when the *next* message is processed. Therefore, when an actor changes its state, it is necessary to create a *new* record to keep track of the new acquaintance bindings: expressions in the current method after the `update` may not see the updated bindings.

6.3 Unrolling RPC Sends

In Section 4.3 we added RPC sends to our basic actor language. However, only asynchronous exchanges are allowed in CHARM. Specifically, when an RPC send is compiled it is necessary to explicitly create continuation methods in the actor definition. Consider an example where an actor of type `Adder`, on receipt of an `add` message, invokes the method `next` of the actor bound to `Get-Res`. It then adds `value` to the returned result:

```
(define-Actor Adder
  (method (add value)
    (+ (rpc-send next Get-Res) value)))
```

We can get the same behavior, with explicit message passing, from an `Adder′` actor:

```
(define-Actor Adder'
   (method (add value)
     (send next' Get-Res add-cont self))
   (method (add-continuation answer)
     (+ answer value)))
```

On receipt of an `add` message, an `Adder′` actor sends a `next′` message to the actor bound to `Get-Res` and requests that the result be sent back to its `add-continuation` method. When the result arrives the `Adder′` actor adds `value` to it as according to the original definition. Notice, that this requires us to propagate the original `value` parameter to the continuation. In addition, if the actor's new behavior depends on the response, all other messages must be ignored by this actor; otherwise, the remembered binding of `value` might get corrupted. Finally, such translations will also require modifying the actor `Get-Res`, since it now must accept an address and method name as parameters and explicitly return its result to the continuation. These translations are done automatically by the compiler. The user can just write code in terms of `rpc-sends`.

7 Results and Future Research

The HAL system is based on CHARM which has been implemented on both shared and nonshared memory machines including Sequent Balance and Symmetry, Encore Multimax, Alliant FX/8, Intel iPSC/2, iPSC/i860 and NCUBE/2[8]. As a result, HAL programs are expected to run on all such machines. HAL programs have been tested on uniprocessor, Encore Multimax and Intel iPSC/2 architectures. Preliminary results on an iPSC/2 indicate that a message send from within HAL which goes across processor boundaries takes approximately ten times the time it takes for a similarly sized message to be sent across processor boundaries from an optimized C program. Creation of an actor, in HAL, on a remote node of the iPSC/2 takes approximately 50% longer than an HAL message send. Note that the primary goal of HAL is high-level programming support; it provides a testbed for ideas in concurrent language design. However, we are currently studying sources of inefficiency.

To exploit internal concurrency, the actor model mandates that updates to local variables only go into effect upon receipt of the next message. This requires the *shadowing* of local variables, if an actor updates a variable, a new state record is created and the new value is placed in the new state so that the old value will be available for the rest of the method. When the actor is finished processing the current method, the system throws out the old state record. This can be rather inefficient as every time a local variable is modified a new state must be allocated. In addition, all of the state variables and system information must be copied over. Using static analysis, one can determine safe, in-place, mutations.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[2] G. Agha, S. Frølund, R. Panwar, and D. Sturman. A linguistic framework for dynamic composition of fault-tolerance protocols. Technical Report UIUC DCS-R-92-1730, University of Illinois at Urbana-Champaign, April 1992.

[3] G. Agha, C. Houck, and R. Panwar. Distributed execution of actor systems. In *Proceedings of Fourth Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, 1991.

[4] P. America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(4):366–411, 1989.

[5] P. America and F. van der Linden. A parallel object-oriented language with inheritance and subtyping. In *OOPSLA '90*, pages 161–168, October 1990.

[6] W. Athas and C. Seitz. Cantor user report version 2.0. Technical Report 5232:TR:86, California Institute of Technology, Pasadena, CA, January 1987.

[7] A. Chien. *Concurrent Aggregates: An Object-Oriented Language for Fine-Grained Message-Passing Machines.* PhD thesis, MIT, July 1990.

[8] W. Fenton, B. Ramkumar, V.A. Saletore, A.B. Sinha, and L.V.Kale. Supporting machine independent programming on diverse parallel architectures. In *Proceedings of the International Conference on Parallel Processing*, pages 193–201, August, 1991.

[9] N. Francez. *Fairness.* Texts and Monographs in Computer Science. Springer-Verlag, 1986.

[10] S. Frølund. Inheritance of synchronization constraints in concurrent object-oriented programming languages. To appear at ECOOP 1992.

[11] C. Houck. Run-time system support for distributed actor programs. Master's thesis, University of Illinois at Urbana-Champaign, May 1992.

[12] N. Hutchinson, R. Raj, A. Black, H. Levy, and E. Jul. The emerald programming language REPORT. Technical Report 87-10-07, University of Washington, October 1987.

[13] S. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[14] D. Kafura, D. Washabaugh, and J. Nelson. Garbage collecton of actors. In *OOPSLA '90*, pages 126–134, October 1990.

[15] L. Kale. *The CHARM(3.0) Programming Language Manual.* University of Illinois, October 1991.

[16] W. LaLonde and J. Pugh. *Inside Smalltalk*, volume 1. Prentice Hall, 1990.

[17] Carl Manning. Acore: The design of a core actor language and its compiler. Master's thesis, MIT, Artificial Intelligence Laboratory, August 1987.

[18] C. Tomlinson, W. Kim, M. Schevel, V. Singh, B. Will, and G. Agha. Rosette: An object oriented concurrent system architecture. *Sigplan Notices*, 24(4):91–93, 1989.

[19] C. Tomlinson and V. Singh. Inheritance and synchronization with enabled-sets. In *OOPSLA*, 1989.

[20] T. Watanabe and A. Yonezawa. *ABCL An Object-Oriened Concurrent System*, chapter Reflection in an Object-Oriented Concurrent Language, pages 45–70. MIT Press, Cambridge, Mass, 1990.

[21] T. Watanabe and A. Yonezawa. A actor-based metalevel architecture for group-wide reflection. In J. W. deBakker, W. P. deRoever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, pages 405–425. Springer-Verlag, 1990. LNCS 489.

[22] P. Wegner. Dimensions of object-based language design. Technical Report CS-87-14, Brown University, July 1987.

[23] A. Yonezawa, editor. *ABCL An Object-Oriented Concurrent System.* MIT Press, Cambridge, Mass., 1990.