Design of an Actor Language for Implicit Parallel Programming

Yariv Aridor **Shimon Cohen** **Amiram Yehudai**

Computer Science Department

Tel-Aviv University, Tel-Aviv 69978, ISRAEL

PH: +972-3-6409299, FAX: +972-3-6409357, E-mail: yariva|amiram@math.tau.ac.il

Abstract

Software tools that support implicit parallel programming hold the key to reducing the complexity of parallel programming and realizing more ubiquitous parallel computation. This paper addresses the topic of implicit concurrent object-oriented programming. It presents a new language, SYMPAL, which unlike most existing COOP languages that provide explicit constructs for concurrency control, is based on a unification of object orientation and pure functional programming with the goals of supporting *implicit programming* and *high efficiency* in massively concurrent object-oriented programming. Extensive experience with SYMPAL applications on a parallel machine indicates that it achieves these goals.

key words: object-oriented programming, concurrency, functional programming, efficiency, implicit programming, Actor-based languages.

1 Introduction

Concurrent Object-Oriented Programming (COOP) has been studied extensively in recent years with the intention of using objects as concurrency units, executed in parallel, for massively parallel programming. Much work has been done on the Actor [1] computational model, which integrates objects and concurrency at the object level. To date, a wide variety of Actor-based languages have been proposed [6, 3, 12], along with techniques their efficient implementation [11, 4, 2]. The languages comprise an extension of the Actor model, with special language constructs for concurrency control (see section 3). Thus, they support *explicit* concurrent object-oriented programming, leaving to expert programmers the tasks of managing communication and synchronization among a vast number of objects in massively parallel programs.

This paper is about simplifying concurrent object-oriented programming. It presents a *practical* programming environment, **SYMPAL**, that incorporates object orientation and pure functional programming with the goals of supporting *implicit programming* and *high efficiency* in massively concurrent object-oriented programming. In **SYMPAL**, a pure functional base language provides an inherently parallel semantics, on top of which extensions to OOP (based on the Actor model) enhance practicality by using mutable objects and notions such as inheritance and encapsulation.

SYMPAL is unique among Actor-based languages in being an inherently parallel language, relying only on compilation technology to produce efficient code. In a more general context, while current forms of implicit programming focus on regular programs and data-parallel algorithms, **SYMPAL** fills the obvious need for an implicit form for task-parallel programming.

This paper focuses on the language design. Detailed descriptions of compile-time and run-time optimization techniques and extensive experience with **SYMPAL** applications on a parallel machine can be found in [2].

2 The SYMPAL Language

SYMPAL is an untyped parallel language composed of a pure functional language, defined with inherently parallel semantics, and extensions to concurrent object-oriented programming, based on the Actor model [1]. This composition of object orientation and functional programming yields the following features:

Improved Linguistic Support for Concurrency

SYMPAL extends support for concurrency in the Actor model beyond one-way asynchronous message-passing.

Implicit Concurrent Object-Oriented Programming

Parallelism in SYMPAL is the default execution mode. Sequencing occurs only when it is necessitated by data dependencies. Specifically, the language is based on (1) asynchronous message-passing, (2) implicit synchronization (i.e., via special code generated automatically by the compiler) to wait for not-yet-ready values of parallel subcomputations (e.g., the reply values of asynchronous messages) and (3) implicit intra-object parallelism (i.e., simultaneous method invocations on the *same* object).

2.1 The Computational Model

The computational model of SYMPAL is Actor [1], which combines objects and concurrency in such a way that:

- Objects are dormant when created and perform actions (defined by methods) only in response to the arrival of messages.
- Messages are handled one at a time, in the order of their arrival (an object has a private message queue to enqueue incoming messages, if necessary).
- An object (actor) responds to a single message (i.e., performs the corresponding method) and then terminates. Thus, every message is handled by a new task. The method may specify a replacement behavior (e.g., assignment of new values to instance variables) that results in a new copy of an object with the same message queue and new behavior. The new object will handle the next message. Consequently, a replacement behavior is only visible in subsequent method invocations. The replacement behavior can also specify what specific type of messages to accept next.

SYMPAL's design extends the Actor model [1] as follows:

- Support of single inheritance. In addition, there is no notion of selecting only specific types of messages to be handled, so inheritance anomalies are avoided [7].
- Replies to messages can be associated with variables.
- Replies to messages are automatically sent back to the sender objects.
- Update of instance variables and intra-object parallelism are supported without creating new copies of objects.

These extensions are described later in the paper.

```
(DEFMETHOD (TreeNode search) (k)
   (IF (= key k) data  ; data,key, left and right are instance variables
       (POR (IF right (SEND right :search k))
            (IF left (SEND left :search k)))))
```

Figure 1: Searching in a Tree of Objects

2.2 The Programming Constructs

SYMPAL inherits its data types, primitive functions, and forms of definitions of classes and methods from COMMON-LISP/Flavors [8]. However, (1) explicit assignment to variables is not allowed and (2) All its constructs are defined to have parallel semantics, so that all expressions can be evaluated in parallel. The constructs are:

function call (F $a_1 \ldots a_n$) or (LET ((v_1 a_1) $\ldots$ (v_n a_n)) *body*)
In both forms, the actual arguments $a_1 \ldots a_n$ and the body of the function are evaluated in parallel.

if-then-else (IF *cond then else*)
The *cond* expression is evaluated first. If its value is non-**nil**, *then* is evaluated: otherwise, *else* is evaluated. `if-then-else` is the synchronization construct.

por (POR $c_1 \ldots c_n$)
All $c_1 \ldots c_n$ expressions are evaluated in parallel. The return value is that of the first expression c_i that terminates with a non-**nil** value, although the computation of the other expressions continues. If all the expressions are evaluated as **nil**, the return value is **nil**.

send (SEND *o msg* $e_1 \ldots e_n$)
A message, *msg* is sent to a target object *o*. The parameters of the message, $e_1 \ldots e_n$, can be any valid SYMPAL expressions. A `send` expression is treated as a function call: (1) all the parameters are evaluated in parallel, (2) the message is sent without waiting for the completion of the evaluation of all its parameters, and (3) its value is the return value of the corresponding method that is invoked.

finally (FINALLY E ((v_1 E_1) $\ldots$ (v_n E_n)))
An object is updated. For a detailed description, see section 2.3.

Figure 1 shows the SYMPAL method for searching, in parallel, in a tree of objects, located by keys. It demonstrates the usefulness of embedding `send` expressions within other expressions and using functional constructs such as `por` to coordinate activities among objects.

2.3 The `finally` Concept

The Actor model [1] includes the `become` primitive to specify a replacement behavior for an object. In practice, a new copy of the object, with the same message queue and new behavior, is created. SYMPAL uses a variation of the `become` construct, named FINALLY, to update instance variables of objects and to control intra-object parallelism without creating new copies of objects. The following is a typical structure of a method body:

```
(defmethod (<method> <class>) (<parameters>)
   (if (<cond>)
       (... ; actions
          (finally (<continuation>) (<assignments>)))
       (... ; actions
          (finally (<continuation>) (<assignments>)) )))
```

The `finally` expressions are used as tail expressions of a method, so exactly *one* `finally` expression is executed during an invocation of a method. Actions may include message-passing and local computation, but not explicit assignment. Once a `finally` expression is reached, it updates the instance variables and unlocks the object. Execution of the current method proceeds with the continuation part, in parallel with the handling of new messages (if any). A `finally` expression has the following syntax:

$$(\text{FINALLY } E \ ((v_1 \ E_1) \ \ldots \ (v_n \ E_n)))$$

where $v_1 \ldots v_n$ are instance variables and E and $E_1 \ldots E_n$ are SYMPAL expressions, excluding `finally` expressions. The semantics is described by the following three steps:

- Each expression E_i is evaluated in parallel. Its future value is bound to the corresponding instance variable v_i. These future values are visible only to subsequent method invocations.[1]
- The object is unlocked. New method invocations are allowed.
- The expression E is evaluated sequentially. Its value is the return value of the method invocation.

In practice, `finally` expressions will be located at the earliest point in a method body where all the new values of variables can be specified.[2]. The object remains locked exactly long enough to guarantee that the next method invocation will use the most recently updated values of the instance variables. This *single-assignment* scheme with the `finally` construct has the following consequences:

1. Intra-object parallelism:
   Intra-object parallelism allows overlapped execution of several method invocations on the same object, while preserving the semantics of the Actor model [1]. This is an improvement over other Actor-based languages [3, 12] in which objects handle messages one at a time, often imposing unnecessary sequencing on subsequent method invocations.
2. Inter-object parallelism:
   Owing to the absence of assignments during the *actions* stage, all the concurrency among the actions is based on the inherent parallelism of the pure base language. This way we provide a language in which parallelism is the default execution mode.

As an example, consider the program in figure 2, which defines the basic operations of **update** and **sum** on a linked list of **listnode** objects. Any tail expression `exp` not wrapped by an explicit `finally` construct (e.g, the **sum** method) is automatically transformed by the compiler into (`finally` *exp* nil). Although most of the source code in figure 2 is written as a sequential program, high levels of

[1]The values of instance variables accessed within the continuation expression and any of the E_i expressions are still the same as at the beginning of the method invocation. In practice, these expressions access (i.e., read from) copies of the referenced variables created automatically by the compiler before the object is updated [2].

[2]Either by the programmer or by compile-time optimizations [2].

```
(DEFCLASS listnode ((mydata nil) (next nil) (mykey nil)))

(DEFMETHOD (listnode update) (key value)
   (IF (= mykey key)
       (FINALLY t
                ((mydata (updatedata mydata value))))
      (IF next
         (SEND next :update key value))))

(DEFMETHOD (listnode sum) ()
   (IF next
       (+ mydata (SEND next :sum))
       mydata))

(DEFUN updatedata (data value)
   ; ..  combine data with value )
```

Figure 2: Parallel Operations on a List

concurrency can be exploited. For example, in **sum**, an object is unlocked to process new messages, while the current method is blocked to wait for the values of **data** (if not yet ready) and the reply to the **sum** message sent to the next object before the addition operation takes place. Another example occurs in the **update** method. The object is unlocked to process new messages in parallel with the computation of a new future value assigned to the **data** variable.

3 Comparison of SYMPAL with other Actor-Based Languages

Consider a program for evaluating a numeric expression represented as a tree of objects. Each internal object represents a numeric operation, and each leaf represents a value. Figure 3 compares a **SYMPAL** version of the program with versions in two ancestor Actor-based languages, Sal and ABCL/1. It includes the definition of one specific **minimum** class, which represents a binary operation to calculate a minimum value among natural numbers. The SAL language [1] is a minimal Actor-based language without any extensions, while ABCL/1 is a more recent language, including explicit constructs for communication and synchronization.

Both the ABCL/1 and the **SYMPAL** versions are much more simple than the Sal program, mainly because variables can be associated with replies to asynchronous messages. Thus, the extra method **get-value** and the two additional instance variables, **father** and **value**, are eliminated.

The ABCL/1 version is based on the `make-future` and `next-value` primitives, associated with asynchronous message-passing, which save its reply value and suspend execution, awaiting the reply value for a message, respectively. With respect to synchronization and communication, the **SYMPAL** version is, actually, an implicit version of the ABCL/1 one, based on asynchronous message-passing and implicit synchronization.

SYMPAL maintains the intra-object parallelism supported in the Actor-model. Thus, both **SYMPAL** and SAL allow a **minimum** object to handle messages such as **is-binary** while waiting for replies to **eval** asynchronous messages. In contrast, ABCL/1 handles messages, by default, in sequence.

ABCL/1 has an explicit notation, "!", for sending back early replies to messages, thus minimizing the delay of a sender object until it receive back a reply. It is currently unclear how to support this feature in the functional context of **SYMPAL**.

```
def minimum (value,right,left,father)
[case operation of
  eval : (customer)
  get-value : (v,sender)
  is-binary : (sender)
end case]
if operation = eval then
   send eval request to right
   send eval request to left
   become minimum(0,right,left,customer)
fi
if operation = get-value then
   let new-value = if value = 0 then v
                   else if value > v then v fi fi
   { if value =/ 0 then
        send <[get-value request with new-value and self]> to father
        become minimum(0,right,left,nil)
     else
        become minimum(new-value,right,left,father)
     fi
   }
fi
if operation = is-binary then
        send <a reply message with a TRUE value> to sender
fi
end def
```

A) SAL version

```
[object minimum
 (state left,right)
 (script
   (=> [:eval]
       (temporary
         [future1:=(make-future)]
         [future2:=(make-future)]
         L R)
       [left <= [:eval] $ future1]
       [right <= [:eval] $ future2]
       [L:=(next-value future1)]
       [R:=(next-value future2)]
       !(if (> L R) R L))
   (=> [:is-binary]
       !T))]
```

B) ABCL/1 version

```
(defclass minimum
   (left, right))

(defmethod (minimum eval) ()
   (finally (let ((L (send left :eval))
                  (R (send right :eval)))
              (if (> L R) R L)))

(defmethod (minimum is-binary) ()
   T)
```

C) SYMPAL version

Figure 3: Comparison of Three Different Actor-based Languages

4 An Example: Nbody simulation

The SYMPAL program in figure 4 is a parallel implementation of Nbody simulation. It computes the motion (positions) of N bodies, attracted by a gravitational force, over a range of time steps. The program has complex communication and synchronization patterns, and thus conveys the flavor of SYMPAL and gives a general impression of its expressive power. The original program appears in [10]. Within the program,

- Each particle is represented by a SYMPAL object. All objects are connected (via the next instance variable) in a ring structure. Each simulation step starts by sending **start-push** messages to activate all the objects in parallel.

- Once an object receives a **start-push** message, it starts to accumulate the forces applied to it by the other objects: it sends a **push** message to its neighbor (pointed to by the next variable), which finally returns the value of the accumulated forces applied to it by the next $\lfloor \frac{N}{2} \rfloor$ objects (starting from its neighbor).

  Meanwhile, the object will receive $\lfloor \frac{N}{2} \rfloor$ **push** messages, sent by the other objects in the ring. Each **push** message contains the position and velocity of the sender so the receiver object can calculate and accumulate the force between it and the sender object. Once all the forces have been accumulated, the object updates its position (the **push** method).

- In every simulation step, calculation of the force between every two objects is based on the values of their positions as computed in the previous step. Thus, a simple protocol, implemented by the **ready** messages, guarantees that a new simulation step starts only after the positions of all the objects have been updated.

- The `parallel` construct is a macro defined on top of the built-in constructs of `function call` and `if-then-else` [2]. It spawns the evaluation of all its arguments in parallel, waiting only for the evaluation of the last one to be completed.

5 Experience

Programming experience with SYMPAL includes several "real" irregular applications such as the SYMPAL compiler itself and Nbody simulations of over 10,000 lines of source code. This experience strengthened our feeling that programming in SYMPAL is as practical and easy as sequential programming. The only difference lies in the single assignment update of instance variables, using the `finally` construct. In addition, its design features of implicit parallelism, intra-object parallelism, and Actor-based semantics, are applicable to a broad class of languages, other than functional ones, such as C++ [2].

SYMPAL has been efficiently implemented on an MIMD machine with eight processors, and on several other machines. Its efficiency is proved by the high performance achieved in the aforementioned SYMPAL applications on these machines [2].

6 Other Languages for Implicit COOP

There are considerable similarities between SYMPAL and a more recent language named UFO [9]. The latter also unifies object orientation and functional paradigms for implicit parallel programming, supporting intra-object parallelism with single-assignment scheme for objects. UFO introduces an alternative approach to the design of an inherently parallel COOP language, based on a data-flow model and a construct for a single assignment inside loops and methods. The compiler detects the point inside a method where the object can be safely unlocked to accept new method invocations (an approach also taken by Hal [6]). More experience is required to compare the pros and cons of every design in terms of simplicity of programming and efficient implementation.

Another implicit COOP language is Mentat [5], a C++-based medium-grained language. However, in comparison with SYMPAL, it is not an Actor language and does not support intra-object parallelism.

```
(DEFCONSTANT N 101)            ; num of particles must be odd !
(DEFCONSTANT N2 (quotient N 2))
(DEFCONSTANT dt 1.0)           ; simulation time step

(DEFCLASS body (index N2) next pos ready (force 0.0) mass vel)

(DEFCLASS rootbody :body  ; subclass of body
  ((steps 0)))

;; constructor methods (missing due to lack of space)

(DEFMETHOD (body start-push) nil
  (parallel (SEND next :start-push)
            (FINALLY t ((force (+ force (SEND next :push n2 0 pos mass)))))))

(DEFMETHOD (rootbody start-push) nil
  (FINALLY t ((force (+ force (SEND next :push n2 0 pos mass))))))

(DEFMETHOD (body push) (hops accforce otherpos othermass)
  (LET* ((f (interforce mass othermass pos otherpos))
         (newforce (+ force f))
         (v (IF (= hops 1) (- accforce f)
                (SEND next :push (- hops 1) (- accforce f) otherpos othermass))))
     (IF (= index 1)
         (parallel (SEND self :ready)
                   (LET ((new-vel (+ vel (* (/ newforce mass) dt))))
                        (FINALLY v
                                 ((vel new-vel)
                                  (pos (+ pos (* new-vel dt)))
                                  (index N2)
                                  (force 0)))))
         (FINALLY v
                  ((force newforce)
                   (index (- index 1)))))))

(DEFMETHOD (body ready) nil
  (parallel (IF ready (SEND next :ready))
            (FINALLY t ((ready (not ready))))))

(DEFMETHOD (rootbody ready) nil
  (parallel (IF ready
                (SEND self :start)
                (SEND next :ready))
            (FINALLY t ((ready (NOT ready))))))

(DEFMETHOD (rootbody start) nil
  (IF (> steps 0)
      (FINALLY (SEND next :start-push)
               ((steps (- steps 1))))))

(DEFUN main (steps)
  (LET ((root (NEW :rootBody N steps))) ; creating a ring of objects
    (SEND root :start)))

(DEFUN interforce (m1 m2 p1 p2)
  (LET* ((r (- p2 p1))
         (r2 (* r r)))
      (/ (* r (* 1.0 (* m1 m2))) (* r2 (abs r))))) ; 1.0 is the gravitation constant
```

Figure 4: N-body simulation

7 Conclusion

We have presented a programming language for implicit concurrent object-oriented programming. The complexity of programming is simplified by a division of labor among the language, compiler and run-time system. The language is inherently parallel, thus facilitating the extraction of large amounts of parallelism. The optimizing compiler and the run-time system are responsible for efficient management of parallelism. The language is simple, yet achieves a high level of expressive power using a minimal set of constructs:

While the work so far has been focused on designing the core language and reaching efficient implementation, future work includes extending the language with features such as synchronization constraints and support for object placement, better modularity (e.g., direct invocation of methods within other methods) and exception handling.

8 Acknowledgments

We thank Mike McDonald of IBM Japan for checking the wording of this paper.

References

[1] G. Agha. *ACTORS: A Model for Concurrent Computation in Distributed Systems.* MIT press, 1986.

[2] Y. Aridor, S. Cohen, and A. Yehudai. Sympal: a software environment for implicit concurrent object-oriented programming. *Object-Oriented Systems*, 4:53–81, 1997.

[3] W. C. Athas. Fine grain concurrent computations. Technical Report TR-87-5242, Computer Science Dept., California Institute of Technology, 1987.

[4] A. Chien, V. Karamcheti, J. Plevyak, and W. Feng. Techniques for efficient execution of fine-grained concurrent programs. In *Proceeding of the Workshop on Language and Compilers for Parallel Machines*, 1992.

[5] A. Grimshaw. Easy-to-use object-oriented parallel processing with MENTAT. *IEEE Computer*, 26(5), 1993.

[6] W. Y. Kim and G. Agha. Compilation of a highly parallel Actor-based language. In *Languages and Compilers for Parallel Computing.* Springer-Verlag, 1992.

[7] S. Matsuoka, K. Taura, and A. Yonezawa. Highly efficient and encapsulated re-use of synchronization code in concurrent object-oriented languages. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 109–126, 1993.

[8] D. Moon. Object-oriented programming with FLAVORS. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 1–8, 1986.

[9] J. Sargeant. Uniting functional and object-oriented programming. In S. Nishio and A. Yonezawa, editors, *Proceeding of the 1st JSSST int'l symposium on object technologies for advanced software*, 1993.

[10] C. L. Seitz. The cosmic cube. *Communications of the ACM*, 28(1):22–33, 1985.

[11] K. Taura, S. Matsuoka, and A. Yonezawa. An efficient implementation scheme of concurrent object-oriented languages on stock multicomputers. In *Proceedings of the Fourth ACM Sigplan Symposium on Principles and Practice of Parallel Programming*, 1993.

[12] A. Yonezawa. *ABCL: An Object-Oriented Concurrent System.* The MIT Press, 1990.