A Simple Model for Concurrent Object-Oriented Programming

Emil Sekerinski

***Abstract*— It has been argued that objects can be naturally thought of as evolving independently and thus concurrently; an object is a natural "unit" of concurrency. Yet, current mainstream object-oriented languages treat concurrency independently of objects: concurrency is expressed in terms of threads that have to be managed separately from objects. We argue for a model for concurrent object-oriented programs in which no such distinction is made. The only syntactic additions needed are extending classes by actions and allowing methods to be guarded. Execution is governed by a simple rule for atomicity. The model allows concurrency to be seamlessly introduced in classes, thus for example allowing concurrency to be introduced in subclasses of a class hierarchy. This permits concurrency to be treated as an implementation issue in the same way as the choice of an algorithm. The model relieves the programmer from having to distinguish between the process and class aspects in software design. We illustrate the model by examples, discuss its rationale, and outline our current implementation.**

***Index Terms*— Object-oriented concurrency, action-based concurrency.**

I. Introduction

A reoccurring theme in the history of "high level" programming languages is how languages relieve the programmer from idiosyncrasies of processors and from details of translation techniques: For example, Fortran translates arbitrarily nested arithmetic expressions by manipulating an expression stack. Algol 60 supports recursion and scoping rules as we know them today. Pascal takes over the memory layout of programmer defined types. Functional and object-oriented languages offer automatic garbage collection of dynamically allocated memory blocks.

It has been argued that objects can be naturally thought of as evolving independently and thus concurrently; an objects is a natural "unit" of concurrency. Yet, current mainstream object-oriented languages treat concurrency independently of objects: concurrency is expressed in terms of processes (or threads) that have to be managed separately from objects. Our goal is to offer the programmer one further relief, namely that from explicitly managing processes of concurrent programs.

Historically processes were introduced for managing physical resources. This is reflected in concurrency theories like process algebras and atomic (shared variable) actions that centre around explicitly declared processes—typically corresponding to resources—and their parallel composition. Eliminating the need for explicitly declaring and composing processes simplifies the design process of object-oriented programs: rather than having the class structure and the process structure as two interdependent design views, we have only one, namely that of the class structure. Concurrency is expressed by adding actions to objects, with the possibility of introducing concurrency in subclasses of a class hierarchy. This permits concurrency to be treated as an implementation issue in the same way as the choice of an algorithm. On the other hand objects can be used to manage resources, very much like processes can, so no expressiveness is lost.

The purpose of this paper is to demonstrate the viability of this "process-less" model of concurrent programming by a series of examples and to argue that an efficient implementation is possible. We start by introducing the language in which the programs are expressed. A formal theory for verification and refinement is outlined in [14]. While in this theory each class is defined in terms of an action system and the whole program as the parallel composition of action systems, no explicit parallel composition operator is present in the language. A prototypical compiler that translates to the Java Virtual Machine is presented in detail in [12]. Here we only sketch the implementation.

Briot et. al. [7] give an useful overview of concurrency in object-oriented programming which we follow to classify our approach. The level of concurrency can be characterized as *quasi-concurrent*, like in ABCL/1 [15], as several method activations may coexist, but at most one is not suspended. This is in contrast to *serial* objects like in POOL [2] that support only one method activation and fully concurrent objects like with Actors [1]. Our objects would be classified as *autonomous* rather than *reactive* as they may be active without receiving a method call; in Java all objects are reactive and autonomous activity is expressed through threads. The acceptance of messages is *implicit* rather than *explicit* as in Ada and POOL, where each object has a *body* that controls entry into the object; condition synchronization is achieved through *guards* instead. The communication between objects is through *synchronous method calls*, as in Ada, POOL, and Java, rather than through message queues as with Actors.

The closest work is the Seuss approach of Misra [13] and OO-action systems of Bosangue, Kok and Sere [5], [6]. We share with these approaches the use of synchronous method calls, the use of guards for condition synchronization, and the use of actions to express autonomous activity. While in Seuss only a fixed number of objects can be declared, we allow dynamic object creation, as in OO-action systems. A difference is how atomicity of actions and methods is guaranteed if they contain multiple method calls. Suppose we have an (unguarded) action $x.m$; $y.n$ and method n of object y is not enabled. In OO-action systems, following the theory of action systems, the whole action is therefore not enabled; thus, if we were to execute $x.m$, we would have to roll back. In Seuss this is solved by allowing a call to a guarded method to be only the first statement in an action or a method. Beside being

inconvenient, this forbids that a unguarded method is refined by a guarded one is a subclass (though Seuss does not consider subtyping). We do not have this restriction, but allow that an action or method "gets stuck" at the point where a method is called. That is, actions and methods are atomic only up to method calls. While our compiler accepts these programs, we note that for verification and refinement programs have to be translated such that method calls appear only as the first statement in methods and actions [14]. The underlying refinement theory of Büchi and Sekerinski [8] assumes this form of programs.

Much of the inspiration and several of our examples come from the $\pi o \beta \lambda$ approach that was initiated by Jones [10], [11], even though $\pi o \beta \lambda$ is defined in terms of the π calculus and our language is defined in terms of action systems. We do not directly support *early return* and *delegate* statements as $\pi o \beta \lambda$ does, but we do allow methods (and actions) to be guarded; $\pi o \beta \lambda$ supports mutual exclusion, but not condition synchronization.

II. A Concurrent Object-Oriented Language

We start by giving the (slightly simplified) formal syntax of the language in extended BNF. The construct $a \mid b$ stands for either a or b, $[a]$ means that a is optional, and $\{a\}$ means that a can be repeated zero or more times:

class ::= **class** *identifier* [**inherit** *identifier*]
  { *attribute* | *initialization* | *method* | *action* } **end**
attribute ::= **attr** *varList*
initialization ::= **initialization** [(*varList*)] *statement*
method ::= **method** *identifier* [(*varList*)] [: *typeList*]
  [**when** *expression* **do**] *statement*
action ::= **action** [*identifier*]
  [**when** *expression* **do**] *statement*
statement ::= **assert** *expression* |
  idList := *exprList* |
  idList :∈ *exprList* |
  identifier.*identifier* [(*exprList*)] |
  idList := *identifier*.*identifier* [(*exprList*)] |
  identifier := **new** *identifier* [(*exprList*)] |
  identifier := *expression* **as** *identifier* |
  begin *statement* { ; *statement* } **end** |
  if *expression* **then** *statement* [**else** *statement*] |
  while *expression* **do** *statement* |
  var *varList* ; *statement* |
  return *exprList*
varList ::= *idList* : *type* { , *idList* : *type* }
typeList ::= *type* { , *type* }
idList ::= *identifier* { , *identifier* }
exprList ::= *expression* { , *expression* }

A class is declared by giving it a name, optionally stating the class being inherited, and then listing all the attributes, initializations, methods, and actions. Initializations have only value parameters, methods may have both value parameters and return a result, and actions don't have parameters. Both methods and actions may optionally have a *guard*, a boolean expression. Actions may be named, though the name does not carry any meaning; while methods can be called using their name, actions cannot be called, they can be executed whenever their guard is true. The assertion statement **assert** b checks whether boolean expression b holds. If it holds, then execution continues, otherwise it aborts. The assignment $x := e$ assigns simultaneously the values of the list e to the list x of variables. The nondeterministic assignment statement $x :\in s$ selects an element of the set s and assigns it to the list x of variables. This statement is not part of the programming language, but is included here for use in abstract programs. A method call $z := c.m(e)$ to object c takes the list e as the value parameters and assigns the result to the list z of variables. If there is no result returned, we simply write $c.m(e)$. The object creation $c :=$ **new** $C(e)$ creates a new object of class C and calls its initialization with value parameter e. The type cast $c := d$ **as** C checks if d is of class C; if so, it assigns d to object c, otherwise it aborts. We do not further define *identifier*, *expression*, and *type*.

III. Reader-Writer Locks

We illustrate the constructs of the language by a series of examples. Consider the problem of ensuring that a resource is either accessed by up to R readers or a single writer. We can do so by objects of the class *ReaderWriter*:

class *ReaderWriter*
  attr n, N : *integer*
  initialization $(R$: *integer*$)$
    $n, N := R, R$
  method *startRead*
    when $n > 0$ **do** $n := n - 1$
  method *startWrite*
    when $n = N$ **do** $n := 0$
  method *endRead*
    $n := n + 1$
  method *endWrite*
    $n := N$
end

The class initialization sets attributes N and n to the maximum number of readers. The class maintains the invariant $0 \leq n \leq N$. Methods *startRead* and *startWrite* are *enabled* if their respective guard is true, otherwise *disabled*. If *rw* is an object created by *rw* := **new** *ReaderWriter*(N), then a typical access of the resource would be *rw.startRead* ; ... ; *rw.endWrite*. The atomicity policy is that all methods and actions are atomic up to method calls. Hence all methods and the initialization of the class *ReaderWriter* are executed atomically, but the calls *rw.startRead* and *rw.endRead* may be suspended. The call *rw.startRead* may continue only when *startRead* becomes enabled, the call *rw.endRead* may continue at any time. We assume here that all attributes are private to an object and all methods are public (our implementation offers a finer control of visibility).

IV. Priority Queue

A priority queue offers a method *add*(e) for storing integer e, a method *remove* for removing the least integer stored so far, and a method *empty* for testing whether the priority

queue is empty. Our implementation is by a linked list of nodes. Elements are stored in attribute *m* in ascending order (duplicates are allowed). Attribute *l* points to the next node or is *nil* at the last object, which does not hold a queue element. An element is added to the priority queue by either storing it in the current node if it is the last one (and creating a new last node), or by depositing it in the current node and enabling an action that will move either the new element or the element of the current node one position down. The minimal element is removed by returning the element of the current node immediately and enabling an action that will move the element of the next node one position up, or set the *l* pointer to *nil* if the node becomes the last one:

```
class PriorityQueue
  attr m, p : integer
  attr l : PriorityQueue
  attr a, r : boolean
  initialization l, a, r := nil, false, false
  method empty : boolean
    when not r do
      return l = nil
  method add(e : integer)
    when not a and not r do
      if l = nil then
        begin m := e ; l := new PriorityQueue end
      else
        begin p := e ; a := true end
  method remove : integer
    when not a and not r do
      begin r := true ; return m end
  action doAdd
    when a do
      begin
        if m < p then l.add(p)
        else begin l.add(m) ; m := p end ;
        a := false
      end
  action doRemove
    when r do
      begin
        if l.empty then l := nil
        else m := l.remove ;
        r := false
      end
end
```

Methods *add* and *remove* are disabled if either *a* is true—a request for adding has been deposited—or *r* is true—a request for removing has been deposited, while *empty* is disabled if *r* is true. We note that *a* and *r* cannot be true at the same time. If *a* is true, then *l* cannot be *nil* and the queue is not empty, hence independently of *a* the test $l = nil$ reflects whether the queue is empty. However, if *r* is true, then the queue may or may not be empty, hence method *empty* has to wait until *r* becomes true.

Actions cannot be called, they can be initiated whenever they are enabled. In principle each object can have one action being executed in parallel with actions of other objects. Thus a priority queue can have at most as many concurrent actions as there are nodes in the queue.

The atomicity policy allows that as soon as control passes to another object, another method call can enter that object, an action of that object can be initiated, or a method or action can resume its suspended execution. Actions *doAdd* and *doRemove* contain calls to another object, hence at those calls exclusive access is dropped. As at the calls $l.add(p)$ and $l.add(m)$ in *doAdd* attribute *a* is false, methods *add* and *remove* and actions *doAdd* and *doRemove* are all disabled while method *empty* is enabled. Thus *empty* can also be called after the call *l.add* leaves the current object and before it returns; for example, that call may get suspended and not return immediately.

Objects are similar to monitors in the sense that both guarantee exclusive access to private data. Method calls to other objects—the equivalent of nested monitor calls—are *open* as the exclusive access to the first object is dropped and only regained when the call returns. By comparison, method calls in Java are *closed* as exclusive access to all objects in the call chain is retained. It is known that closed calls allow less concurrency and are more prone to deadlocks, see e.g. [3] for a discussion. On the other hand, open calls require the class invariant to be established or all methods and all actions of that object to be disabled before a call to another object. While the second approach has the same effect as closed calls our point is that the programmer can chose either extreme or some approach in between by selectively disabling methods.

We note that compared to traditional monitors (and to some extend to Java), there are no condition variables, no *signal* and *wait* operations, and no processes as explicit language constructs—their role is taken over by guarded methods and actions.

V. Leaf-Oriented Trees

The next example is about parallelizing operations on sets. The implementation is by leaf-oriented trees, i.e. trees in which the internal nodes contain only guides and the elements are stored in the leaves. Insertion either creates two new leaves or only deposits an element in an internal node. Each node has an action that would eventually move the deposited element one level closer to its final position. This action needs to hold a lock only on the current node and one of its children. Thus insertions can proceed in parallel in different parts of the tree. The methods *add* and *has* are guarded in order to prevent possible overtaking:

```
class Tree
  attr root : Node
  initialization root := nil
  method add(x : integer)
    if root = nil then root := new Node(x)
    else root.add(x)
  method has(x : integer) : boolean
    if root = nil then return false
    else return root.has(x)
end
```

```
class Node
  attr key, p : integer
  attr left, right : Node
  attr a : boolean
  initialization (x : integer)
    key, left, right, a := x, nil, nil, false
  method add(x : integer)
    when not a do
      if left ≠ nil then a, p := true, x
      else if x < key then
        left, right, key := new Node(x), new Node(key), x
      else if x > key then
        left, right := new Node(key), new Node(x)
  method has(x : integer) : boolean
    when not a do
      if left = nil then return x = key
      else if x ≤ key then return left.has(x)
      else return right.has(x)
  action addToChild
    when a do
      begin
        if p ≤ key then left.add(p)
        else right.add(p) ;
        a := false
      end
end
```

The class *Node* maintains the local invariant $(left = nil) = (right = nil)$ and the global invariant $(left \neq nil) \Rightarrow (left.key \leq key) \wedge (right.key > key)$.

VI. Observer Pattern

The last example is the observer design pattern [9], expressed as an abstract program. The pattern allows that all observers of one subject perform their *update* methods in parallel:

```
class Observer
  attr sub : Subject
  initialization (s : Subject)
    begin sub := s ; s.attach(this) end
  method update ...
end

class Subject
  attr a, n : set of Observer
  initialization a, n := {}, {}
  method attach(o : Observer)
    a := a ∪ {o}
  method notifyAll
    n := a
  action notifyOne
    when n ≠ {} do
      var o : Observers ;
      begin o :∈ n ; n := n − {o} ; o.update end
end
```

As soon as execution of the action *notifyOneObserver* in a subject *s* reaches the call *o.update*, control is passed to object *o* and another activity in *s* may be initiated or may resume. In particular, the action *notifyOneObserver* may be initiated again, as long as *notifyObs* is not empty, i.e. some observers have not been notified. Thus at most as many *notifyOneObserver* actions are initiated as there are observers and all notified observers can proceed concurrently. New observers can be added at any time and will be updated after the next call to *notify*.

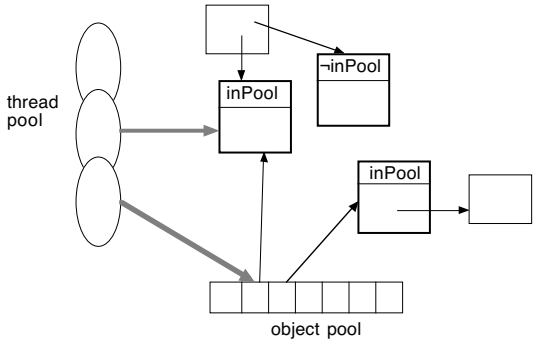

Fig. 1. Illustration of the implementation. Boxes with the *inPool* attribute represent active objects, the other boxes represent passive objects. A thin arrow between objects represents a reference, a thick arrows from a thread to an object represents a reference with a lock.

VII. Implementation

In order to test our ideas, we have developed a prototypical compiler for our language, see [12] for details. The compiler currently translates to the Java Virtual Machine. We sketch the principles of the implementation, see Fig. 1 for an illustration. The implementation relies on the restriction that method and action guards may refer only to attributes of the object itself and may not contain method calls. An object that has guarded methods is called a *guarded object*. An object that has actions is called an *active object*, otherwise a *passive object*. An active object that has at least one enabled action is called an *enabled object*, otherwise a *disabled object*.

At runtime a *thread pool* and an *object pool* are maintained. The object pool is initially empty. When an active object is created, a pointer to it is placed in the object pool (only active objects are ever placed in the object pool). Each active object has an extra boolean attribute *inPool* indicating whether a pointer to it is in the object pool. Threads request a reference to an active object from object pool and evaluate the action guards. If the object is disabled, the thread resets the *inPool* attribute and removes it from the object pool. If the object is enabled, the thread executes an enabled action and leaves the object in the object pool. Each thread obtains a lock to an object when entering one of its methods or actions and releases the lock when exiting the method or action. The lock is also released at a call to another object and obtained again at re-entry from the call. If a guarded method is called the guard is evaluated and the thread waits if the guard is false. At the exit from a guarded object all waiting threads are notified to reevaluate the guards.

Fairness among the actions of an object is ensured by evaluating the guards in a cyclical order. This is done with one additional attribute for the index of the last evaluated action guard in every active object. The object pool is implemented as a dynamic array. Fairness among the objects is ensured by retrieving active objects in a cyclical order. The object pool grows and shrinks like a stack: new objects are added at the end and when an object is retrieved, its position is filled with the last object. Hence adding objects and retrieving objects take constant time. Active objects are garbage collected like passive objects, i.e. when there is no reference from any other object and no reference from the object pool. With this scheme, there is no need for a thread to search in the object pool or even in the heap for an enabled object and garbage collection is not affected by the presence of active objects.

The object structure effectively helps to control the evaluation of guards. A guard can be thought of as an *await* statement preceding the body. In the absence of any syntactic constraints, the statement **await** *cond* requires repeated evaluation of *cond* after some delay. To reduce resource contention, a *binary exponential back-off protocol* can be employed that starts with a random delay and doubles it after each failure, similar to the Ethernet protocol. In the presented scheme no delays are employed. Action guards are initially evaluated once when a thread is searching for an action to execute. Method guards are initially evaluated once when a method is called. Both action and method guards are reevaluated only after another thread has exited an action or method of the object and thus possibly affected the guards.

The memory overhead is that every active object requires one bit for the *inPool* attribute, one integer for the index to the last evaluated action guard, and one pointer in the object pool. The number of object threads equals to the size of the object pool array—thus every enabled object will eventually get its turn. In our current implementation the object pool array can only grow but not shrink, and so can the number of threads.

VIII. Discussion

The viability of the proposed approach crucially depends on the efficiency of the implementation. Our main focus so far was developing the underlying theory [8], [14]. An early experimental implementation for a similar, but more restricted language, was presented in [4] and our current one in [12]. The compiler currently does not carry out any optimizations like eliminating synchronization statements when not needed and detecting which guards do not need reevaluation after specific exits. There remain the topic of future work. Hence we have so far not conducted any timing experiments. However, we did measure the size of the object and thread pool. In programs like the ones presented, it turns out that typically the object and thread pool size is significantly smaller than the number of active objects (i.e. an order of magnitude or more). Hence in these examples thread management does not constitute a bottleneck. This gives us some reassurance that highly efficient implementations are possible.

Acknowledgment

The author would like to thank the reviewers for their careful reading and constructive comments.

References

[1] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. Series in Artificial Intelligence. MIT Press, Cambridge, MA, 1986.

[2] Pierre America. Pool-T: A parallel object-oriented language. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, Computer Systems Series. MIT Press, Cambridge, MA, 1987.

[3] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addision-Wesley, 2000.

[4] Ralph Back, Martin Büchi, and Emil Sekerinski. Action-based concurrency and synchronization for objects. In T. Rus and M. Bertran, editors, *Transformation-Based Reactive System Development, Fourth AMAST Workshop on Real-Time Systems, Concurrent, and Distributed Software*, Lecture Notes in Computer Science 1231, pages 248–262, Palma, Mallorca, Spain, 1997. Springer-Verlag.

[5] Marcello M. Bonsangue, Joost N. Kok, and Kaisa Sere. An approach to object-orientation in action systems. In *Mathematics of Program Construction*, Lecture Notes in Computer Science 1422, Marstrand, Sweden, 1998. Springer-Verlag.

[6] Marcello M. Bonsangue, Joost N. Kok, and Kaisa Sere. Developing object-based distributed systems. In P. Ciancarini, A. Fantechi, and R. Gorrieri, editors, *3rd IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'99)*, pages 19–34. Kluwer, 1999.

[7] Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Lohr. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3):291–329, 1998.

[8] Martin Büchi and Emil Sekerinski. A foundation for refining concurrent objects. *Fundamenta Informaticae*, 44(1):25–61, 2000.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[10] Cliff B. Jones. An object-based design method for concurrent programs. Technical report, University of Manchester, Department of Computer Science, December 1992.

[11] Cliff B. Jones. Accomodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8(2):105–122, March 1996.

[12] Kevin Lou. *A Compiler for an Action-Based Object-Oriented Programming Language*. Master's thesis, McMaster University, 2003.

[13] Jayadev Misra. A simple, object-based view of multiprogramming. *Formal Methods in System Design*, 20(1):23–45, 2002.

[14] Emil Sekerinski. Concurrent object-oriented programs: From specification to code. In *First International Symposium on Formal Methods for Components and Objects, FMCO 02*, Lecture Notes in Computer Science, Leiden, The Netherlands, 2003, to appear. Springer-Verlag.

[15] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *ACM Conference on Object Oriented Programming Systems, Languages and Applications*, ACM SIGPLAN Notices, Vol 21, No 11, pages 258–268, 1986.