

Concurrent Aggregates (CA)¹

Andrew A. Chien and William J. Dally

andrew@ai.mit.edu billd@ai.mit.edu

Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract

To program massively concurrent MIMD machines, programmers need tools for managing complexity. One important tool that has been used in the sequential programming world is hierarchies of abstractions. Unfortunately, most concurrent object-oriented languages construct hierarchical abstractions from objects that serialize – serializing the abstractions. In machines with tens of thousands of processors, unnecessary serialization of this sort can cause significant loss of concurrency.

Concurrent Aggregates (CA) is an object-oriented language that allows programmers to build unserialized hierarchies of abstractions by using aggregates. An aggregate in CA is a homogeneous collection of objects (called representatives) that are grouped together and may be referenced by a single aggregate name. Aggregates are integrated into the object model, allowing them to be used wherever an object could be used. Concurrent Aggregates also incorporates several innovative language features that facilitate programming with aggregates. Intra-aggregate addressing aids cooperation between parts of an aggregate. Delegation allows programmers to compose an concurrent aggregate behavior from a number of objects or aggregates. Messages in CA are first class objects that can be used to create message handling abstractions (they handle messages as data). Such abstractions facilitate concurrent operations on aggregates. Continuations are also first class objects. In addition, programmers can construct continuations and use them just like system continuations. User constructed continuations can implement synchronization structures such as a barrier synchronization.

¹ The research described in this paper was supported in part by the Defense Advanced Research Projects Agency and monitored by the Office of Naval Research under contracts N00014-88K-0738 and N00014-87K-0825, in part by an NSF Presidential Young Investigator Award with matching funds from GE Corporation and IBM Corporation, and in part by an Analog Devices Fellowship.

1 Introduction

Parallel programming is important because it offers a means for solving larger and more complex problems. Several research groups are building multiprocessors with thousands of powerful processors. The aggregate computing potential of these machines will exceed 500 billion instructions per second [8, 4]. If parallel programs can take advantage of the massive hardware concurrency we can afford to build (and make work reliably), such programs will be able to solve more complex problems, including many that are economically or practically impossible to solve with existing computer systems.

We are concerned with utilizing the computing potential of large ensembles of processors. At least two major obstacles stand between us and that goal.² First, programming should be relatively easy – it must not be so difficult that programs take years to construct and debug. As in large sequential programs, it must be possible to use abstraction (and hierarchies of abstractions) to relegate details to the appropriate level in a program. In fact, the importance of abstraction techniques is perhaps greater in parallel systems as the presence of non-deterministic behavior can complicate debugging. Second, the language must allow us to express sufficient concurrency to utilize the machine. In an ensemble of 10,000 – 100,000 processors, unnecessary serialization may dramatically reduce the achievable performance.³

Most concurrent object oriented languages serialize hierarchical abstractions. In languages such as ACORE [15], ABCL/1 [16], CANTOR [3] and POOL-T [2], hierarchical abstractions (abstractions built from other abstractions) are built from single objects. Their objects may accept only one message at a time – resulting in serialized abstractions.⁴ Multiple levels of abstraction can

²There are several others such as load balancing and concurrent I/O but we will not discuss them here.

³A simple argument based on Amdahl's law [1] confirms this.

⁴For example, in the Actor model, the serial message reception order is the actor's lifeline. This assumption is reflected in the notion that an actor's behavior can change at the reception of each message. In fact, in any language that assumes that an object resides on only one node, that processing node becomes a

result in greatly diminished concurrency – even if each level only causes a tiny amount of serialization. This leaves programmers with the choice of reduced concurrency or working without useful levels of abstraction. Going without these levels of abstraction makes programs more difficult to write, understand, and debug.

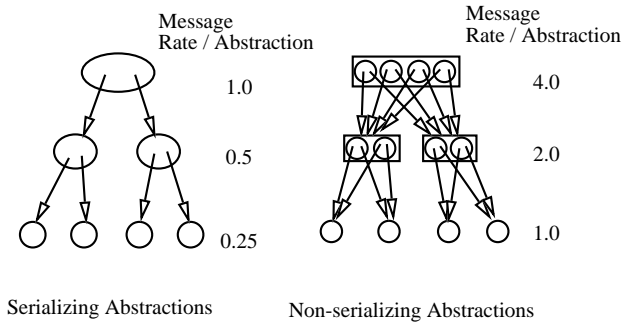


Figure 1: Maximum Message Rates in Abstraction Hierarchies

Concurrent Aggregates allows programmers to build hierarchical abstractions without serialization as shown in Figure 1. The message rates shown are normalized to the maximum message reception rate of a single object. CA programmers can build hierarchical abstractions from aggregates. Each aggregate is multi-access and therefore can receive many messages simultaneously. By using appropriately sized aggregates in the upper levels of hierarchy, we can increase the message rate for lower levels in the hierarchy.

Concurrent Aggregates incorporates four additional important concepts that help programmers to construct abstractions from collections of objects and aggregates.

- Intra-aggregate Addressing
- Delegation
- First Class Messages
- First Class Continuations

Intra-aggregate addressing allows representatives of an aggregate to compute the names of other parts of the aggregate. This facilitates object cooperation in implementing abstractions by allowing representatives to pass messages to each other conveniently. For efficient communication, internal aggregate names can be used to link representatives directly to other objects in the same or in other aggregates. Externally, the aggregate serialization point for its objects.

can be manipulated with a single name – in a manner identical to an ordinary single object. This enables aggregates and single objects to be used interchangeably.

Delegation can be used to piece together (structurally compose) one aggregate’s behavior from the behavior of others. In CA, an aggregate can delegate the handling of one or more messages to another aggregate. An aggregate can also delegate the handling of all messages not handled locally to another aggregate – implementing more traditional form of delegation [13].

Delegation may facilitate concurrency by enabling programmers to distribute message handling responsibility over a collection of aggregates.

First class messages allow programmers to write message manipulation abstractions. Such abstractions can be used to implement control structures that perform message reordering or implement data parallel⁵ operations on aggregates. Such aggregate operations are an important source of concurrency in our programs.

CA treats continuations as first class objects. This enables programs to code synchronizing abstractions such as futures. Further, ordinary objects or aggregates can be used as continuations (assuming they handle the reply message) making it possible for programmers to construct complex continuation structures such as a barrier. These structures can be cleanly factored from the remainder of the program. These four features – intra-aggregate addressing, delegation, first class messages and first class continuations – aid a programmer in constructing complex, concurrent aggregate behaviors.

1.1 Background

We are developing Concurrent Aggregates to program fine-grain concurrent computers such as the J-machine [8, 7]. These machines are characterized by massive concurrency ($\approx 10^5$ nodes), fast communication networks [9] (latency of $< 2\mu s$ – roughly 20 instruction times), small local memories ($\approx 64K$ words), and support for fine-grain computation (hardware support for message passing, fast context switching, fast task creation and dispatch). Each of the nodes executes instructions from its own local memory (i.e. this is an MIMD machine). Although there is a global shared address space, there is no shared memory. Nodes communicate via asynchronous message passing. Machines such as the J-machine have tremendous performance potential if we can develop effective ways of programming them.

⁵This differs from the usage of the term “Data Parallel” in the context of SIMD machines. We mean the parallelism arising from operations on large sets of data, be it heterogeneous or homogeneous. No global synchronization is implied.

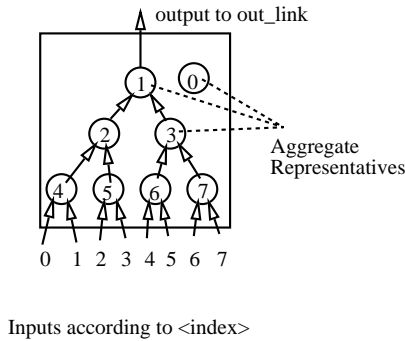


Figure 2: A Combining Tree of size 8

Concurrent Aggregates is being developed in the Concurrent VLSI Architecture Group at MIT. CA was motivated by the complexity of writing large programs in existing concurrent object oriented languages. Mechanisms for building aggregates in these languages all lead to serialization of message reception. This added serialization is unacceptable for programming massively concurrent hardware. Like Concurrent Smalltalk (CST) [6, 10, 11], concurrency in CA is derived from asynchronous messages sends and synchronization through context futures. Aggregates in CA are similar to distributed objects in Concurrent Smalltalk (CST). However, CST lacks the specific support for programming with aggregates incorporated in CA. To support aggregates, CA provides delegation, first class messages and programmer constructed continuations. These features are described in detail in Section 3. CA programmers can use these tools for constructing, composing, and interconnecting aggregates.

1.2 Overview

The remainder of this paper describes the Concurrent Aggregates language and highlights its novel features. Section 2 presents a simple example program. Section 3 presents the language and describes its novel features. Where practical, we present code fragments illustrating language features. Section 4 presents the current status of our CA implementation and experimentation. Finally, Section 5 summarizes the contributions of CA and suggests directions for further study. A more complete description of the Concurrent Aggregates language can be found in [5].

2 An Example

To introduce the idea of a concurrent aggregate, we describe a combining tree. Conceptually, a combining tree accepts a number of values – one from each of its leaves – and combines them with some associative binary operation. The resulting value is produced at the top of the tree. Our combining tree aggregate accepts a number of messages of the form: (`combine <combiningtree> <selector> <value> <index>`). The `<selector>` specifies the binary operation that should be used to combine values and should be the same in all `combine` messages. `value` is the value to be combined and `index` specifies which leaf originated the `combine` message.

In our implementation of a combining tree, `combine` messages are directed to an arbitrary representative by the runtime system. The `combine` handler converts this to an `internal_combine` message and uses the `index` argument to send it to the appropriate tree node. Within the `combine_tree` aggregate, `internal_combine` messages propagate up the tree until the representative at the root sends an `internal_combine` message to the output, `out_link`. The internal links of the combining tree used to propagate the `internal_combine` messages are shown in Figure 2. The CA code for the combining tree is shown in Figure 3. To illustrate program behavior, values for a number of CA forms are presented below:

```
(global console) => (reply os:console)

(new combine_tree 8 (global console))
=> (reply aggregate:7234)

(let ((ctree(new combine_tree 8 (global console))))
  (forall index from 0 below 8
    (do (combine ctree + 1 index)))
  (reply done))
=> (reply done)
(internal_combine + 8)
```

The first expression shows an access to the global variable, `console`. The `reply` message contains a descriptor for the console. The second expression sends a `new` message to the operating system, which replies with a descriptor for the newly created `combine_tree` aggregate. The third expression puts it all together; showing the creation, use and replies from the combining tree. Evaluating expression three results in two response messages: the result of the expression, `done`, and the message from the root of the combining tree, `internal_combine`.

```

;;
;; Combining Tree Abstraction
;;
(aggregate combine_tree myparent value state
  (parameters number_leaves out_link)
  (initial number_leaves
    (forall index from 1 below groupsize
      (set_myparent
        (sibling group index)
        (sibling group (/ index 2)))
        (set_state (sibling group index) 0))
      (set_myparent (sibling group 1) out_link)))

(handler combine_tree combine (selector val index)
  (let ((rep_index (+ (/ groupsize 2)
                      (/ index 2))))
    (forward (internal_combine
              (sibling group rep_index) selector val))))

(handler combine_tree internal_combine (selector val)
  (if (= 0 (state self))
      (conc (set_value self val)
            (set_state self 1))
      (let ((combined_val
              (selector val (value self))))
        (set_state self 0)
        (do (internal_combine
              (myparent self) selector combined_val)
            )))

```

Figure 3: An Implementation of a Combining Tree

3 Programming with Concurrent Aggregates

In this section, we present the syntax of the Concurrent Aggregates language. For each set of forms, we present the syntax and then an explanation of those forms. The first set, class and aggregate definitions, are the basis of object structure in CA. The syntax for aggregate and class definitions is given below.

```

(aggregate <aglname> instance-variable*
  (parameters param-name+)
  (initial <aggsz>
    exp+))

(class <cname> instance-variable*
  (parameters param-name+)
  (initial exp+))

(global <gname> initial-exp)

```

CA programs consist of a series of aggregate, class and global definitions. Aggregate definitions specify the state (variables) and initialization for the aggregate being defined. The `parameters` clause is used to parameterize

the aggregate being created. The actual parameters are extracted from the `new` message used to create the aggregate. For example, one common use of parameters is to determine the number of representatives to create for an aggregate. Upon creation, the aggregate `initial` code is executed by aggregate representative 0. When the `initial` code returns, the aggregate descriptor is returned. The `aggsz` in the `initial` expression specifies the number of representatives in this instance of the aggregate. `aggsz` may either be a parameter of the aggregate or a constant integer. The `parameters` clause is optional. The `initial` clause is required for aggregates as it contains a specification of the number of representatives desired. Class definitions do not contain an `aggsz` term in the `initial` code, so `initial` forms are optional in class declarations. Otherwise, class definitions follow the same structure as aggregate definitions. Global definitions specify the name of a global variable and an expression to be computed to define its initial value.

For example,

```

(aggregate counter count
  (parameters number_reps icount)
  (initial number_reps
    (forall index from 1 below number_reps
      (set_count (sibling group index) icount))
    (set_count self icount)))

```

defines the `counter` class to have one instance variable, `count`, and two parameters. The first parameter, `number_reps` is used in the `initial` form to determine the number of representatives in the counter aggregate. The second parameter is used to initialize the local state. Upon creation, each `counter` aggregate initializes the `count` variable in each of its representatives to the value of `icount`.

Method and handler definitions plus delegations form the basis of object and aggregate behavior in CA. We give the syntax and explanation for these forms below.

```

(handler <aglname> <messagename> (arg*) exp+)
(method <cname> <messagename> (arg*) exp+)
(delegate <agg-or-cname> <messagename>
  instance-variable)

```

Aggregate and object behavior is specified by the definition of handlers and methods. Handlers are used to define code executed by aggregates, while methods are used to define code for objects. Behavior can also be specified by delegations. Delegations pass responsibility for handling a particular message to another object or aggregate. A delegation of the “:rest” message causes

all messages not handled locally (by a method or handler) to be delegated. “:rest” delegation is similar to that proposed in [13].

Method and handler bodies contain control constructs and message sends. The syntax for both of these is given below.

```
(seq exp+)
(conc exp+)
(if cond-exp exp0)
(if cond-exp exp0 exp1)
(let ((var exp+) exp+)
  (forall <varname> from exp0 below exp1
    exp+)
  (<selector> receiver args*) ;; message send
  (reply exp)
```

`seq` and `conc` specify sequential and concurrent execution of the expressions they contain. `let` binds variables to values, but does not require variable value evaluation to be complete before the `let` body begins execution. Synchronization for variable values is done via *context futures* [6]. In method and handler bodies, any use of values not yet available causes execution to suspend until the required value is available. `forall` causes multiple executions of expressions in `exp+`. Different iterations may be, but are not guaranteed to be concurrent. The message send transmits a message named `selector` to the `receiver` with `args*` as arguments. `reply` sends a reply with the value of `exp` to the current continuation.

```
(count self)
(set_count self 23)
```

Instance variables are accessed and modified with message passing syntax. For example, the above expressions in a handler for the counter aggregate would read the value of `count` and set it to 23, respectively.

3.1 Aggregation and Naming

An aggregate in CA is a homogeneous collection of objects (called representatives) which are grouped together and may be referenced by a single aggregate name. Messages sent to the aggregate are directed to arbitrary representatives as shown in Figure 4. Since this *one-to-one-of-many* direction is performed by the runtime system (itself multi-access), aggregates are multi-access and do not introduce serialization – each representative can receive messages concurrently.

The representatives of an aggregate can communicate by sending messages to one another. As shown in the

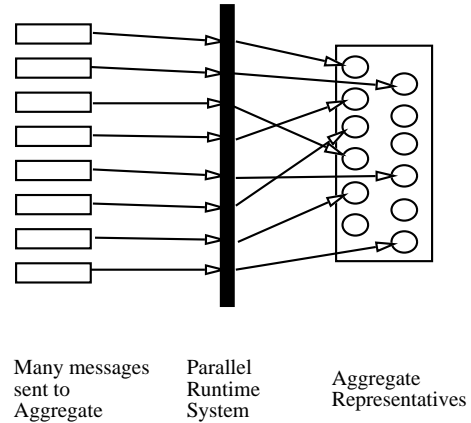


Figure 4: Aggregate to Sibling Translation

example of Figure 3, representatives can address one another using the form `(sibling group <index>)` that allows a representative to get the names of its siblings. For example, `(sibling group 0)` would return the name of the 0th representative in an aggregate. The representatives in an aggregate have indices from 0 to `groupsize-1`. `groupsize` is a pseudo-variable which contains the number of representatives in an aggregate. Sibling names are ordinary object names – allowing direct connection to aggregate representatives when performance is critical.

```
...
(initial number_leaves
  (forall index from 1 below groupsize
    (setmyparent (sibling group index)
      (sibling group (/ index 2)))
    (set_state (sibling group index) 0))
  (setmyparent (sibling group 1) out_link))
...

```

The above code fragment from the combining tree in Figure 3 demonstrates direct connections between the representatives in an aggregate. The `myparent` links created here point directly to a representative’s parent in the combining tree. This allows `internal_combine` messages to avoid forwarding. If necessary, direct links can be made to representatives from outside of the aggregate.

3.2 Delegation and Message Handling

When a message with selector `X` is received, the handlers and delegations defined for the aggregate determine how the message is handled as shown in Figure

```

;; first, include these other abstraction definitions
(load "counter.ca")
(load "loan_officer.ca")
(load "tellers.ca")
;;
;; Bank abstraction
;;
(aggregate bank tellers loan_officer manager
  (parameters number_reps bankmanager)
  (initial number_reps
    (let ((cash (new counter 1000)))
      (let ((tellers_agg (new tellers (/ number_reps 2) cash))
            (loan_off (new loan_officer cash)))
        (forall index from 1 below groupsize
          (let ((curr_sib (sibling group index)))
            (set_tellers curr_sib tellers_agg)
            (set_manager curr_sib bankmanager)
            (set_loan_officer curr_sib loan_off)))
        (set_tellers self tellers_agg)
        (set_manager self bankmanager)
        (set_loan_officer self loan_off))))))

(delegate bank deposit tellers)
(delegate bank withdraw tellers)
(delegate bank balance tellers)
(delegate bank loan_request loan_officer)
(delegate bank loan_payment loan_officer)

;; Anything unusual handled by manager
(delegate bank :rest manager)

```

Figure 5: A Bank Aggregate

6. If a method or handler is defined for **X**, that code is invoked. If a delegation is defined for **X**, the message is sent to the delegation target. Method definitions and delegations for a class should not conflict (i.e., there should not be a method definition and a delegation for the same selector) . If neither method nor delegation is defined and a “:rest” delegation is defined for **X**, then the message is sent to the “:rest” delegation’s target. Methods and handlers are defined to handle messages in objects and aggregates respectively. Handlers are very similar to methods, but CA uses a different reserved word to remind programmers they are dealing with an aggregate.

Delegation allows programmers to construct aggregate behavior incrementally. Specific methods may be used to extend or modify the message interface of an aggregate while a “:rest” delegation specifies a default receiver object for all other messages. For instance, consider the bank aggregate example shown in Figure 5. The behavior of the bank aggregate is made up of a number of parts. The **deposit**, **withdraw**, and **balance**

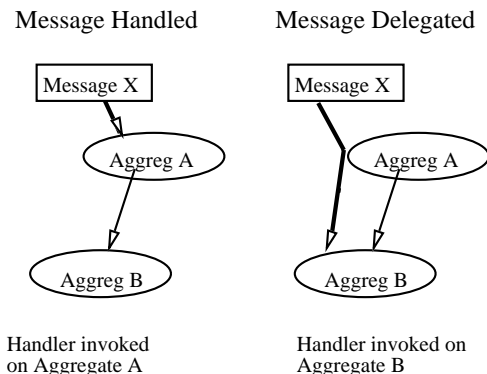


Figure 6: Message Handling and Delegation

messages are handled by the tellers aggregate. The **loan_request** and **loan_payment** messages are handled by the loan officer and anything else is handled by the manager. The **initial** code initializes each represen-

<u>Bank Message Interface</u>	<u>Message Implementor</u>
<i>deposit</i>	<i>tellers</i>
<i>withdraw</i>	<i>tellers</i>
<i>balance</i>	<i>tellers</i>
<i>loan_request</i>	<i>loan_officer</i>
<i>loan_payment</i>	<i>loan_officer</i>
<i>:rest (anything else)</i>	<i>manager</i>

Figure 7: Bank Aggregate Message Interface

tative’s state to hold references to the objects or aggregates that really handle the messages. For example, in each **bank** representative, the **tellers** variable contains a reference to a shared **tellers** aggregate (definition not shown). By enabling programmers to piece behaviors together, CA allows programmers to distribute message handling responsibility over a number of objects – potentially increasing concurrency.

3.3 Explicit Message Handling

CA allows the explicit creation, manipulation and modification of messages. The syntax and descriptions for forms to do so are given below.

```
(send <message>)
(send <message> <receiver>)
(msg_at <message> <index>)
(msg_atput <message> <value> <index>)
(message <application-exp>)
```

Concurrent Aggregates treats selectors and messages as first class objects. By first class, we mean that CA programs can manipulate selectors and messages explicitly by storing, copying, and sending them. Several language forms in CA allow programmers to send, create, and modify messages. For example, **send** with one argument transmits a message. With two arguments, **send** changes the message’s receiver and then transmits it. **msg_at** and **msg_atput** give array style access to a message.

Programmers can get references to messages via the **message** form or the **msg** pseudo-variable. The **message** form returns as its value the message represented in the **application-exp**. For example, **(message (test a b))** returns a message with selector **test**, receiver **a**, and one argument **b**. Messages created with the **message** form have null continuations. Methods and

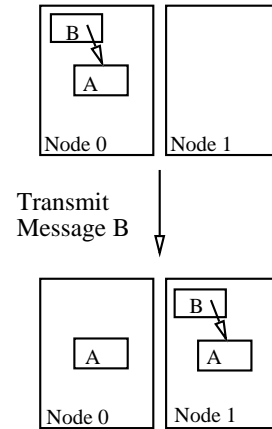


Figure 8: Messages are *by-value* parameters

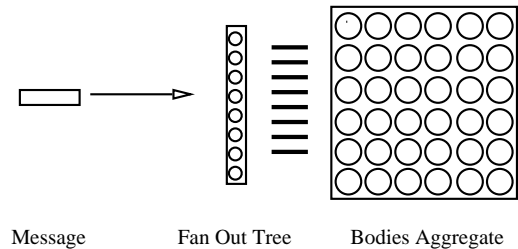


Figure 9: An aggregate operation – move bodies

handlers can refer to their invoking message with the pseudo-variable, **msg**.

Messages are by-value parameters: they are copied at method invocation as shown in Figure 8.⁶ This means that if message B contains a reference to message A the sending of message B causes A to be copied. After the sending of B, the new copy of A is located on the same node as the receiver of message B. An important consequence of this is that messages are quite often local objects.

First class messages can be used to implement operations on collections – exploiting “data parallelism.” For instance, in an N-body interaction simulation, first class messages can be used to implement position update for a collection of bodies. Figure 9 depicts such a scenario. The CA code required to implement the fan out **tree** aggregate is shown in Figure 10. A single message sent to the fan out tree causes all bodies in the collection to

⁶This copying is only one level deep. For instance, if a method invocation had a message argument **M1** and **M1** contained a reference to message **M2**, message **M2** would not be copied.

```

;; first, include the range abstraction
(load "range.ca")
;;
;; A fanout tree
;;
(aggregate tree
  (parameters treesize) (initial treesize))
(handler tree fan_out (mess bodies-agg range)
  (if (> (size range) 2)
    (let ((lrange (split_range range)))
      (do (fan_out group mess bodies-agg lrange))
      (do (fan_out group mess bodies-agg range))
      ;; range has been side-effected
      (forall index from (low range) below (high range)
        (send mess (sibling bodies-agg index))))))
;;
;; A bodies aggregate
;;
(aggregate bodies xpos ypos xvel yvel mass
  (parameters psize)
  (initial psize))

... other handlers ...

;; Sample Usage
;;
;; (fan_out <tree>
;; (message (move <doesn't-matter> 10)) <bodies>)

```

Figure 10: Abstract Implementation of operations on a Collection

receive `move` messages with the current time step. Each time a `fan_out` message is executed, the message parameter, `mess`, is copied. Each `fan_out` message is processed by a representative determined by the runtime system. Thus, when we reach the leaves of the fan out tree, there is one copy of the message for each representative of the bodies aggregate. The last stage of the fan out tree transforms the messages (by writing the appropriate representative's name into message's destination field) and sends them.

The `tree` aggregate demonstrates an interesting use of aggregates – as computational bandwidth. The fan out function of the tree makes no use of state in the `tree` aggregate. The representatives are only used as computation sites. This means the fan out tree can be used by many fan out operations simultaneously. One could also use the size of the fan out tree aggregate to limit the fan out rate for the tree.

3.4 Continuations

The Concurrent Aggregates language allows programs to manipulate continuations as first class objects. Continuations may be stored, copied, and have messages sent to them. Programs can get references to continuations by using `msg_at` on a message or accessing the `requester` pseudo-variable. `requester` contains the value of the current continuation. The `do` and `forward` forms shown below can be used to modify continuation values for messages.

```

(do <application-exp>)
(do <application-exp> <continuation>)
(forward <application-exp>)

```

CA contains language constructs to perform some common operations on continuations. With one argument, the `do` form performs asynchronous sends (passing a null continuation, so there is no reply). A second argument can be used to specify a continuation. This is especially useful in CA as programs can substitute ordinary objects and aggregates for continuations. Tail forwarding [10] (the distributed machine analog of tail recursion optimization) can be done using the `forward` form. Of course, (`forward <exp>`) is equivalent to (`do <exp> requester`).

```

(load "pair.ca")
;;
;; A future Object
;;
(class future tag val deferred
  (initial (set_tag self 0)
            (set_deferred self 0)))
(method future value ()
  (if (= 0 (tag self))
    (set_deferred self
      (new pair requester (deferred self))
      (reply (val self)))))
(method future set_value (value)
  (set_val self value)
  (set_tag self 1)
  (seq
    (do (forward_replies (deferred self) value))
    (set_deferred self 0))
  (reply value))

```

Figure 11: Code for a Future Object

As has been observed in sequential languages, allowing programmer manipulation of continuations can simplify programs. For example, with first class continuations, one can implement future objects [12] as shown

```

;;
;; A barrier synchronization abstraction
;;
(class barrier count maxcount next_message
  (parameters imaxcount inext)
  (initial
    (set_count self 0)
    (set_maxcount self imaxcount)
    (set_next_message self inext)))
(method barrier reply (val)
  (seq (set_count self (+ val (count self)))
    (if (= (count self) (maxcount self))
      (send (next_message self))))))

```

Figure 12: A Barrier Synchronization Object

in Figure 11. The future object understands two messages: **value**, which returns the value of the future, and **set_value**, which defines the future’s value. When empty (value is undefined), a future object pushes the continuations of all **value** requests onto a list. When the future receives a **set_value** message, replies are sent to each continuation on the list. Subsequent **value** messages receive immediate responses.

The future object consists of three instance variables: **tag** – holds full or empty status, **val** – the future’s value, and **deferred** – a linked list of continuations. When the value is received, it is stored in **val**, **tag** is modified to indicate full, and the value is sent to all of the continuations on the list. The **pair** object definition is not shown.

3.5 Objects as Continuations

CA allows objects or aggregates to be used as continuations. In a concurrent message passing language, a continuation is simply an object expecting a **reply** message. Based on this observation, we decided to allow CA programs to substitute any object or aggregate that handles the **reply** message for a continuation. This feature facilitates the construction of complex synchronization structures – now continuations can perform any user program computation. This makes it possible to factor the synchronization code out of programs and into abstractions. This allows the remaining program code to be written without regard for the synchronization context in which it is being used. For example, barrier synchronization (set synchronization) can be implemented as shown in Figure 12. Each **reply** message to a barrier object is processed by the **reply** method. After **count** has reached **maxcount**, all elements of the set have arrived and the barrier is complete. Upon completion, the barrier abstraction sends an arbitrary message,

```

(load "pair.ca")
;;
;; a Race abstraction
;;
(class race val tag deferred
  (initial (set_tag self 0)
    (set_deferred self 0)))
(method race reply (value)
  (if (= 0 (tag self))
    (seq (set_val self value)
      (set_tag self 1)
      (do (forward_replies
        (deferred self) value))
        (set_deferred self 0))))
(method race value ()
  (if (= 0 (tag self))
    (set_deferred self
      (new pair requester (deferred self)))
    (reply (val self))))

```

Figure 13: A Race object for speculative concurrency

next_message, which starts the next stage of the computation. The computations being synchronized need not know that they are being barrier synchronized – they obviously send **reply** messages to their continuations. Of course, the barrier abstraction implementation could be concurrent – collecting the reply messages in a combining tree.

It is also possible to construct many other complex synchronization structures. As an example, we present code for a “race” object in Figure 13. A race object was introduced in the Actor model as a way to describe speculative concurrency [14]. A number of computations are started and the race object takes on the value returned by the first to complete. **value** messages received before any of the computations has returned are deferred – the race object also performs future style synchronization. Later replies are discarded. A code fragment that uses the race object to get the first answer from three different solution strategies is shown below.

```

(let ((race_object (new race)))
  (do (strategy1 a b c) race_object)
  (do (strategy2 a b c) race_object)
  (do (strategy3 a b c) race_object)
  (reply (value race_object)))

```

4 Current Status

We have fully defined the Concurrent Aggregates language. Our CA compiler produces C++ code which is

linked with a parallel simulation system. The simulation system implements the CA runtime system in C++ and allows us to simulate CA programs on an abstract fine-grain message passing machine. By using C++, the output of the compiler will be portable to many machines. The CA compiler has been functioning since July 1989 and has been released within our lab since September 1989. Numerous small program kernels and algorithms have been coded, compiled, and executed. We are currently writing several large application programs and plan to report on the effectiveness of CA in managing program complexity.

5 Conclusion

Concurrent Aggregates allows programming with hierarchies of abstractions. In many other languages, the object model causes abstractions be serialized. However, aggregates in CA are multi-access – allowing them to be used to construct unserialized abstractions. The Concurrent Aggregates language accomplishes the integration of aggregates into the concurrent object-oriented model. Aggregates are defined, initialized, interconnected and composed in the same manner as objects. Of course, an aggregate can be used anywhere an object can. Concurrent abstractions built from aggregates can be used to manage the complexity of large scale MIMD concurrent programs.

Concurrent Aggregates incorporates several novel language features that facilitate programming with aggregates. First, CA allows object behavior to be defined via delegation. To our knowledge, this is the first use of delegation in a concurrent object-oriented language. Second, messages are first class objects in CA. This allows programmers to write message manipulating abstractions. This idea is present in object-oriented languages that incorporate some form of computational reflection. CA incorporates first class messages in a way that allows them to be used to build control structures, such as a fan out tree. Third, not only are continuations first class, programs in CA can construct continuations. With such continuations, programmers can separate code from the synchronization context in which it is being used. This allows CA programmers to modularize synchronization structures.

There are still many challenges remaining in the area of concurrent object oriented programming with aggregates. We list a few of the questions yet to be answered. Are the abstraction tools in CA enough to manage complexity in large programs? How powerful is allowing user constructed continuations? Is it only useful for complex synchronization structures or something more?

What is the implementation cost of having messages as first class objects? What about resource management issues, will they affect the language?

References

- [1] G. M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *AFIPS Conference Proceedings*, pages 483–5. AFIPS, 1967.
- [2] P. America. Inheritance and Subtyping in a Parallel Object-Oriented Language. In *Proceedings of ECOOP*, pages 234–42. Springer-Verlag, June 1987.
- [3] William C. Athas. *Fine Grain Concurrent Computations*. PhD thesis, California Institute of Technology, 1987. 5242:TR:87.
- [4] William C. Athas and Charles L. Seitz. Multicomputers: Message-Passing Concurrent Computers. *IEEE Computer*, pages 9–24, August 1988.
- [5] Andrew A. Chien. CA Language Report, Version 1.0. MIT Concurrent VLSI Architecture Group Memo 26, August 1989.
- [6] William J. Dally. *A VLSI Architecture for Concurrent Data Structures*. Kluwer Academic Publishers, Boston, Mass., 1987.
- [7] William J. Dally and et. al. Architecture of a Message-Driven Processor. In *Proceedings of the 14th ACM/IEEE Symposium on Computer Architecture*, pages 189–196. IEEE, June 1987.
- [8] William J. Dally and et.al. The J-Machine: A Fine-Grain Concurrent Computer. In *Proceedings of the IFIPS Conference*, 1989.
- [9] William J. Dally and Paul Song. Design of a Self-Timed VLSI Multicomputer Communication Controller. In *Proceedings of the International Conference on Computer Design*, pages 230–4. IEEE Computer Society, 1987.
- [10] W. Horwat, A. Chien, and W. Dally. Experience with CST: Programming and Implementation. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 101–9. ACM SIGPLAN, ACM Press, 1989.
- [11] Waldemar Horwat. Concurrent Smalltalk on the Message-Driven Processor. Master's thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, June 1989.
- [12] R. Halstead Jr. Parallel Symbolic Computing. *IEEE Computer*, pages 35–43, August 1986.
- [13] Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. In *Proceedings of OOPSLA '86*, pages 214–23. ACM SIGPLAN, ACM Press, 1986.

- [14] Henry Lieberman. Concurrent Object Oriented Programming in ACT 1. In Aki Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 9–36. MIT Press, 1987.
- [15] Carl R. Manning. ACORE: The Design of a Core Actor Language and its Compiler. Master’s thesis, Massachusetts Institute of Technology, August 1987.
- [16] A. Yonezawa, E. Shibayama, T. Takada, and Y. Honda. Object-Oriented Concurrent Programming – Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1. In Aki Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 55–89. MIT Press, 1987.