Massachusetts Institute of Technology | Microsystems Research Center | Cambridge Massachusetts 02139 | Room 39-321 Telephone (617) 253-8138

VLSI Memo No. 88-450
June 1988

OBJECT-ORIENTED CONCURRENT PROGRAMMING IN CST

William J. Dally and Andrew A. Chien

Abstract

CST is a programming language based on Smalltalk-80 that supports concurrency using locks, asynchronous messages, and distributed objects. Distributed objects have their state distributed across many nodes of a machine, but are referred to by a single name. Distributed objects are capable of processing many messages simultaneously and can be used to efficiently connect together large collections of objects. They can be used to construct a number of useful abstractions for concurrency. This paper describes the CST language, gives examples of its use, and discusses an initial implementation.

Acknowledgements

This research was supported in part by the Defense Advanced Research Projects Agency under contract nos. N00014-80-C-0622 and N00014-85-K-0124, an NSF Presidential Young Investigators Award, General Electric Corporation, and an Analog Devices Fellowship.

Author Information

Daily and Chien: Department of Electrical Engineering and Computer Science, Artificial Intelligence Lab, MIT, Cambridge, MA 02139. Daily: Room NE43-417, (617) 253-6043; Chien: Room NE43-411, (617) 253-8572.

Object-Oriented Concurrent Programming in CST [1]

William J. Dally and Andrew A. Chien
Artificial Intelligence Laboratory and
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract

CST is a programming language based on Smalltalk-80 that supports concurrency using locks, asynchronous messages, and distributed objects. Distributed objects have their state distributed across many nodes of a machine, but are referred to by a single name. Distributed objects are capable of processing many messages simultaneously and can be used to efficiently connect together large collections of objects. They can be used to construct a number of useful abstractions for concurrency. This paper describes the CST language, gives examples of its use, and discusses an initial implementation.

1 Introduction

This paper describes CST, an object-oriented concurrent programming language based on Smalltalk-80 [8]. CST adds three extensions to sequential Smalltalk. First, messages are asynchronous. Several messages can be sent concurrently without waiting for a reply. Second, several methods may access an object concurrently, locks are provided for concurrency control. Finally, CST allows the programmer to describe distributed objects, objects with a single name but distributed state. They can be used to construct abstractions for concurrency.

CST is being developed as part of the J-Machine project at MIT [3], [2]. The J-Machine is a fine-grain concurrent computer. It efficiently executes tasks with a grain size of 10 instructions and supports a global virtual address space. This machine requires a programming system that allows programmers to concisely describe programs with method-level concurrency and that facilitates the development of abstractions for concurrency.

Object-oriented programming meets the first of these goals by introducing a discipline into message passing. Each expression implies a message send. Each message invokes a new process. Each receive is implicit. The global address space of object identifiers eliminates the need to refer to node numbers and process IDs. The programmer does not have to insert send and receive statements into the program, keep track of process IDs, and perform bookkeeping to determine which objects are local and which are remote.

For example, a CST program[2] that calculates Fibonacci numbers using double recursion is shown in Figure 1. Nowhere in the program does the programmer explicitly specify a send or receive, and no node numbers or process IDs are mentioned. Yet, as shown in Figure 2 the program exhibits a great deal of concurrency. Making message-passing implicit in the language simplifies programming and makes it easier to describe fine-grain concurrency.

An object-oriented language also encourages locality. Operations on an object happen at the object, not from a distance

```
(Method Integer fib () ()
        (if (< self 2)
            (reply 1)
            (reply (+ (fib (- self 1))
                      (fib (- self 2))))))

(fib 15) ==> 987
```

Figure 1: A CST program to calculate Fibonacci numbers using double recursion.

[1]The research described in this paper was supported in part by the Defense Advanced Research Projects Agency under contracts N00014-80-C-0622 and N00014-86-K-0124, in part by a National Science Foundation Presidential Young Investigator Award with matching funds from General Electric Corporation, and in part by an Analog Devices Fellowship.

[2]This program is in *profs* CST, a dialect that has a syntax resembling LISP. *Full* CST [4] has a syntax closer to that of Smalltalk-80.

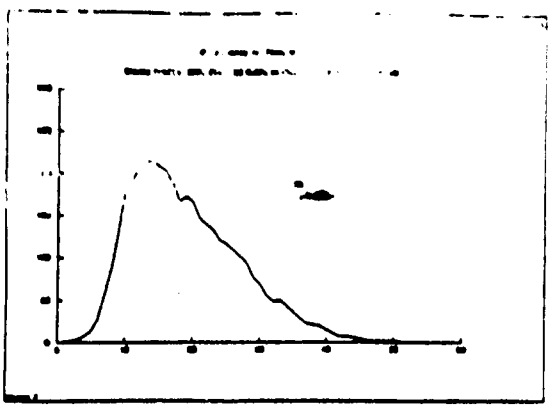

Figure 2: Concurrency profile of Fibonacci program. The plot shows the number of active tasks during each message interval.

using a remote process.

CST facilitates the construction of concurrency abstractions by providing distributed objects: objects with a single name whose state is distributed across the nodes of a concurrent computer. The one to many naming of distributed objects along with their ability to process many messages simultaneously allows them to efficiently connect together large numbers of objects. Distributing the name of a single distributed queue to sets of producer and consumer objects, for example, connects many producers to many consumers without a bottleneck.

Background

The development of Concurrent Smalltalk was motivated by dissatisfaction with process-based concurrent programming using sends and receives [7]. Many of the ideas were borrowed from actor languages [1].

Another language named Concurrent Smalltalk has been developed at Keio University in Japan [3]. This language also allows message sending to be asynchronous but does not include the ability to describe distributed objects.

Outline

The remainder of this paper describes CST and its implementation. Section 2 presents the abstract syntax of the language, describes the primitive types and operations supported by the language, and gives a simple example. In Section 3, we describe distributed objects. We discuss the mechanisms provided to address distributed objects and their constituents as well as several examples. Section 4 describes an implementation of CST.

2 Concurrent Smalltalk

Top-Level Expressions

A CST program consists of a number of top-level expressions. Top level forms include declarations of program and data as well as executable expressions. Linking of programs (the resolution from selectors to methods) is done dynamically.

```
<top-exp> :=    (Global <global-variables>) |
                (Constant <constant-name> <value>) |
                (Class <class-name> (<superclasses>)
                       <instance-vars>) |
                (Method <class-name> <method-name>
                        (<formals>) (<locals>)
                        <expressions>) |
                <expression>
```

Globals and Constants

Globals and constant declarations define names in the environment. These names are visible in all programs, unless shadowed by a instance, argument, or local variable name. The global declaration simply defines the name. Its value remains unbound. The constant declaration defines the name and binds the name to the specified value.

Classes

Objects are defined by specifying classes. Objects of a particular class have the same instance variables and understand the same messages. A class may inherit variables and methods from one or more superclasses. For example:

```
(class transistor (circuitelement)
  source drain gate type size state)
```

defines a class, `transistor`, that inherits the properties of class `circuitelement` and adds six instance variables. This means that methods for the class `transistor` can access all the instance variables of class `circuitelement` as well as those defined in their own class definition. Methods defined for class `circuitelement` are also inherited. Instance variables in the class definition may hide (shadow) those defined in the superclasses if they have the same name. The same kind of shadowing is allowed for selectors (method names).

Methods

The behavior of a class of objects is defined in terms of the messages they understand. For each message, a method is executed. That execution may send additional messages, modify the object state, modify the object behavior, and create new objects.

Methods consist of a header and a body. The header specifies class, selector, arguments, and locals. The body consists of

one or more expressions. For example:

```
(method transistor vgs () ()
  (reply (- (voltage gate) (voltage source))))
```

defines a method for class transistor with selector vgs. The two empty lists indicate that there are no arguments and no local variables. The keyword reply sends the result of the following expression back to the sender of the vgs message. In the absence of a reply keyword, the method replies with the value of the last expression. If the programmer wishes to suppress the reply, he can use the (exit) form which causes the method to terminate without a reply.

Messages are sent implicitly. Every expression conceptually involves sending a message to an object. Of course, commonly occurring special cases, like adding two local integers, will be optimized to eliminate the send. For example, (voltage gate), sends the message voltage to gate. (+ a b) sends the message + with argument b to object a. If a and b are both local integers, this can be optimized into a single add instruction.

Each expression consists of a selector, a receiver, and zero or more arguments. Identifiers must be one of: constant, global variable, argument, local variable, or instance variable. For example, in the method below,

```
(Global Vt)
(Method Transistor foo (vx) (vy)
  (reply (frob vx gate vy Vt 5)))
```

The expression (frob vx gate vy vt 5) consists of a selector, frob, a receiver vx and four arguments: gate, vy, Vt, and 5. In the sending method, foo, frob and 5 are constants, vx is an argument, gate is an instance variable, vy is a local variable, and Vt is a global.

Subexpressions may be executed concurrently and are sequenced only by data dependence. For example in the following expression

```
(- (voltage gate) (voltage source))
```

The two voltage messages will be sent concurrently and the - message will be sent when both replies have been received. The only way to serialize subexpression evaluation is to assign intermediate results to local variables.

A complete list of CST expressions is shown below:

```
<exps> ::= <exp>*
<exp> ::=
    <name> |
    (<selector> <receiver-exp> <argument-exp>*) |
    (send <selector-exp> <receiver-exp>
          <argument-exp>*) |
    (value <exp>) |
    (set <name> <exp>) |
    (cset <name> <exp>) |
    (msg <node> <selector> <receiver> <actuals>) |
    (forward <continuation> <selector>
             <receiver> <args>) |
    (reply <exp>) |
    (block (<formals>) (<locals>) <exps>) |
    (if <exp> <exp> <exp>) |
    (begin <exps>) |
    (exit)
```

Run Time Environment

As in other dynamically linked systems such as Smalltalk and Lisp, we can think of much of the run time environment as programs that are "preloaded" into the environment before the user program is executed. The primitive classes and operations listed below are treated as such.

```
Atomic classes: INTEGER, SYMBOL, FLOAT, BOOLEAN
Composite types: arrays
Arithmetic (integer and float): + - * /
        min max mod rem < > = <= >= !=
Boolean: if and or not
Symbol: eq
Array access: at at.put
Distributed Object: co
Misc: new touch
```

Many primitive operations are defined on integers, floats, booleans, and symbols. These typical operations are found in many languages. The less intuitive primitives are for arrays – arrays are allocated on single nodes (this does not prevent us from building distributed arrays using distributed objects). Values are written using the at.put message and read using the at message. new is a predefined message which allows us to create objects.

Touch simply allows us to require synchronization. Touch requires that its arguments be available for reading. This allows us to control where suspension can occur.

An Example Program

This program integrates a function over the specified interval using a trapezoidal approximation in each subinterval. The number of times the interval is subdivided and hence the subinterval size is determined by epsilon, the maximum interval allowed to be approximated as a trapezoid.

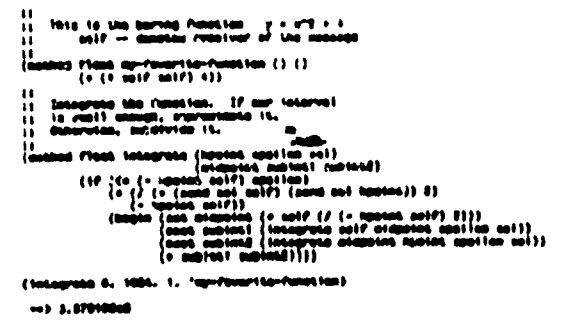

Figure 3: A CST program to integrate a function using double recursion.

The my-favorite-function and integrate methods are both declared for the class of floats. my-favorite-function takes only one parameter, the value for which we want to calculate the function. This argument is the implicit self argument, as the float of interest receives the function selector and calculates the value of the function. Integrate takes the low point of the interval (self), the high point of the interval (hpoint), the interval size (epsilon), and the selector for the function (sel) as arguments. The concurrency profile for an execution of integrate is given in Figure 4.

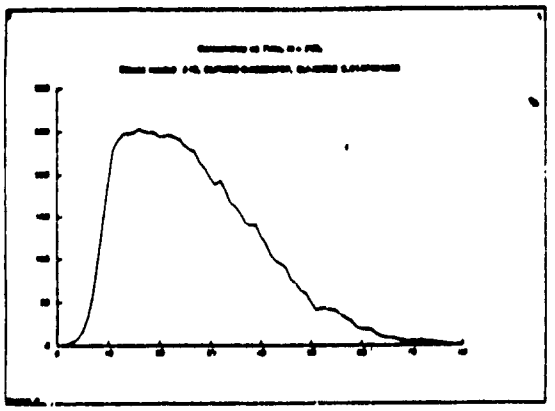

Figure 4: Concurrency profile of Integrate program. The plot shows the number of active tasks during each message interval.

The spawning of parallel computations occurs at the two successive cast operations. This construct allows the method execution to proceed without waiting for a response to the integrate message. We could have produced similar behavior without the use of the locals subint1 and subint2. We introduce the locals to illustrate the cast construct. The results from the concurrent sends get written into subint1 and subint2. The send operation in the last line of the method (send of the + selector) requires both subinterval values and thus causes the method execution to suspend until both results have been returned.[3]

3 Distributed Objects

CST programs exhibit parallelism between objects, that is many objects may be actively processing messages simultaneously. However, ordinary objects can only process a single message at a time. CST relaxes this restriction with Distributed objects (DOs). Distributed objects are made up of multiple representatives (constituent objects) that can each accept messages independently. The distributed object has a name (Distributed object ID or DID) and all other objects send messages to this name when they wish to use the DO.

Messages sent to the DO are received by one and only one constituent object (CO). Which constituent receives the message is left unspecified in the language. A clever implementation might send the messages to the closest constituent whereas a simpler implementation might send the messages to a random constituent. The state of distributed object is typically distributed over the constituents so response to an external request often requires the passing of messages amongst the constituents before the reply to the request is sent. No locking is performed on the distributed object as a whole. This means that the programmer must ensure the consistency of the distributed object.

Support for Distributed Objects

CST includes two constructs to support distributed objects. For DO creation, we add an argument for the new selector - the number of constituents desired in this DO. In order to pass messages within the object, each constituent object must be able to address each of the other constituents. This implemented with the special selector co. Each distributed object can use this selector, the special instance variable group (a reference to the DO), and an index to address a constituent. For example, (co group 5) refers to the 5th constituent of a distributed object. Each constituent also has access to its own index and the number of constituents in the entire distributed object. Thus a description of a distributed object might look something like the example shown in Figure 5.

In the example of the distributed array, we would create a usable array with two steps. First we construct the distributed object using the new form. The example in Figure 5 creates a distributed object with 256 constituents. After the DO is created, we must initialize in a way that is appropriate for the distributed array. We do so by sending it an init

[3] This behavior is analogous to "futures" [6], however, this limited usage of them allows us to implement them more efficiently.

message (also defined in Figure 5). This initialization sets each constituent up with an private array of the appropriate number of elements. For example, if we wanted a distarray of 512 elements, in this case each constituent would have a private array of two elements. This initialization is done in a tree recursive fashion and therefore takes $O(lg(n))$ time.

The mapping of the distarray elements onto the private arrays is done by the at and at:put methods. Each constituent is responsible for a contiguous range of the distarray elements. Any requests received by a constituent are first checked to see if they are within the local CO's jurisdiction. If they are not, they are forwarded to the appropriate CO. If they are, we handle the request locally. This is a particularly simple example because each constituent is wholly responsible for his subrange and need not negotiate with other constituents before modifying his local state.

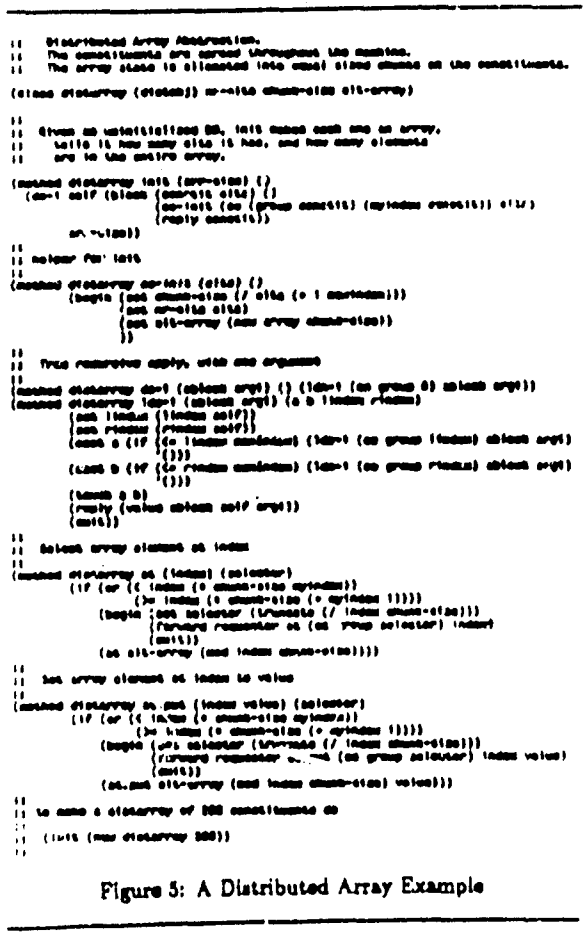

Figure 5: A Distributed Array Example

Other Distributed Object Examples

Distributed objects are of great utility in building large objects on a fine grain machines. In the J-machine, we restrict ordinary objects to fit within the memory of single node. This places a strict limit on object size. With distributed objects, we only require that a constituent of the DO fit on a single node. Some useful examples for distributed objects are dictionaries, distributed arrays, sets, queues, and priority queues.

4 Implementation

A simple programming environment for CST has been implemented on the a Symbolics 3600 system. This environment includes a compiler (which does incremental compilation), a simulator, and plotting facilities. The compiler accepts a *Prefix* CST program and produces intermediate codes (Icodes). These Icodes can be executed on the simulator yielding concurrency profiles, processor utilizations, dynamic Icode mixes, and various other kinds of information. The Icodes can be used as a source for various compiler back-ends. One such back-end, currently under development in the Concurrent VLSI Architecture Group at MIT, targets the Message Driven Processor's [3] instruction set.

Compiler

The Icodes generated by the compiler are similar to the bytecodes of the Smalltalk-80 system. However, we only require eleven different Icodes, far fewer than in the Smalltalk-80 system. Icodes are significantly higher level than instructions in a typical machine. Typical Icodes are MOVE, SEND, JUMP, REPLY, FALSEJUMP, etc. In Figure 6, we give the Icodes for the Fibonacci routine in Figure 1.

One can see that the Icodes clearly reflect the structure of the CST program. The portion preceding (label 0) is clearly identifiable as the conditional test and one of its arms. The remainder of the Icodes are the other arm of the conditional.

The compiler performs two optimizations specific to concurrent programs: tail forwarding and code reordering. Tail

```
(send (temp 0) < self (const 2))
(falsejump (temp 0) 0)
(move (temp 1) (const 1))
(reply (temp 1))
(label 0)
(csend (temp 2) - self (const 1))
(csend (temp 3) fib (temp 2))
(csend (temp 4) - self (const 2))
(csend (temp 5) fib (temp 4))
(send (temp 1) + (temp 3) (temp 5))
(reply (temp 1))
```

Figure 6: Intermediate Codes for fib program.

forwarding is similar to tail recursion. When the value returned from a method is the reply from a called method,

the reply is short-circuited by having the called method reply directly to the original sender. Code reordering moves message sends earlier in a code block to generate additional concurrency.

Simulator

The simulator interprets Icode programs output by the compiler. However, CST programs require several kinds of run time support (hardware and low level OS services) in order to run. Services required by CST (and provided by the simulator) include implementation of virtual address space (ID to node translation, ID to segment address translation), synchronization support (dispatch on message arrival and traps on futures), addressing support for distributed objects, and primitive object placement and migration support. The kernel of the operating system for the J-machine, JOSS [9] provides similar services.

The simulator allows us to study the macroscopic behavior of CST programs. For example, the concurrency profile in Figure 2 was generated by the Icode simulator. Such a system enables us to study issues of placement, concurrency control, partitioning and other resource management problems without worrying about the irrelevant architectural detail.

5 Conclusion

In this paper, we have presented a new language, Concurrent Smalltalk, that is designed for concurrency. Specific support for concurrency includes locks, distributed objects, and asynchonous message passing.

Distributed Objects represent a significant innovation in programming parallel machines. We refer to the constituents of a distributed object with a single name, but the implementation of the object is with many constituents. This different perspective allows easy use of distributed object by outside programs while allowing the exploitation of internal distributed object concurrency.

We have described an implementation of a CST system. This programming environment includes a compiler, simulator, and statistics collection package. This set of tools allows us to experiment with new constructs and implementation techniques for the language.

Concurrent Smalltalk requires significant run time support in order to execute efficiently. Such support has been implemented in the simulator and in JOSS, the operating system for the J-machine.

References

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

[2] Dally, W. J., "Fine-Grain Message-Passing Concurrent Computers," *these proceedings.*

[3] Dally, William J. et.al., "Architecture of a Message-Driven Processor," *Proceedings of the 14th ACM/IEEE Symposium on Computer Architecture*, June 1987, pp. 189-196.

[4] Dally, William J., *A VLSI Architecture for Concurrent Data Structures*, Kluwer, Boston, MA, 1987.

[5] Yokote, Yasuhiko and Tokoro, Mario, "Concurrent Programming in ConcurrentSmalltalk," *Object-Oriented Concurrent Programming*, Yonezawa and Tokoro eds., MIT Press, Cambridge, MA, 1987, pp. 129-158.

[6] Halstead, Robert H., "Parallel Symbolic Computation," *IEEE Computer*, Vol. 19, No. 8, Aug. 1986, pp. 35-43.

[7] Su, Wen-King, Faucette, Reese, and Seitz, Charles L., *C Programmer's Guide to the Cosmic Cube*, Technical Report 5203:TR:85, Dept. of Computer Science, California Institute of Technology, September 1985.

[8] Goldberg, Adele and Robson, David, *Smalltalk-80, The Language and its Implementation*, Addison Wesley, Reading MA, 1984.

[9] Totty, Brian, "An Operating System Kernel for the Jellybean Machine," *MIT Concurrent VLSI Architecture Memo*, 1987.