A Verified Integration of Parallel Programming Paradigms in CC++*

Paolo A.G. Sivilotti
Computer Science Department
California Institute of Technology
Pasadena, California, 91125, USA
paolo@vlsi.cs.caltech.edu

Abstract

CC++ is an object-oriented parallel programming language that uses parallel composition, atomic functions, and single-assignment variables to express concurrency. We show that this programming paradigm is equivalent to several traditional imperative communication and synchronization models, namely monitors and asynchronous channels. Furthermore, the object-oriented nature of CC++ provides an ideal framework for integrating these paradigms. We specify, implement, and formally verify a collection of libraries that integrates these traditional models with CC++.

1 Introduction

Many communication and synchronization primitives have been developed to express parallelism in user programs. Most of these constructs can be shown to be equivalent in expressive power. This equivalence result is sometimes seen to support the view that it is sufficient to provide any one paradigm in a parallel programming language. Our work is motivated by the observation that the integration of more than one paradigm in a programming language is desirable, despite the equivalence result. That is, a programmer might wish to use a particular set of primitives based on concerns other than expressiveness. For example:

Flexibility: one particular methodology is often most appropriate for a given problem instance.

Facility: a single problem instance might lend itself to a decomposition in which different components are expressed using different paradigms.

Familiarity: programmers often become comfortable with a particular methodology. Their experience with that methodology is an important asset, and they should have the opportunity to use it.

Portability: it should be possible to easily upgrade existing code to a new language without paying the penalty of excessive code modification.

Therefore, it is useful for a parallel programming language to support and integrate a variety of styles and methodologies.

Single-assignment variables can be used as a mechanism for communication and synchronization among concurrent processes. Single-assignment variables can be seen as delayed-assignment constants. Such a variable is initially undefined, and it can be written to (or *defined*) at most once. A subsequent attempt to write to the variable is a run-time error. If a process attempts to read a single-assignment variable that has not yet been defined, that process suspends until the variable becomes defined [4, Section 2.1].

We demonstrate that this single-assignment paradigm is consistent with traditional imperative communication and synchronization schemes, and furthermore that CC++ [3, 8] — an object-oriented language based on C++ that incorporates single-assignment variables, atomic functions, and parallel composition — can support and seamlessly integrate these schemes with object-orientation. This support and integration is demonstrated by providing a collection of generic libraries whose correctness is formally verified.

Libraries, rather than language extensions, are used to provide this support, so that new compilers need not be developed. Support and integration at this level is facilitated by the powerful abstraction techniques, such as generic classes and inheritance, provided by C++. Hence, object-orientation plays a key role in the effectiveness of the integration.

Section 3 of this paper is a brief summary of the CC++ programming language. Section 4 describes a monitor library—its specification, an outline of the

*This work was supported in part by Air Force Office of Scientific Research grant AFOSR-91-0070.

design and formal verification, and an example of its use. Section 5 contains a similar description of an asynchronous channel library. Section 6 concludes and summarizes the results.

2 Related Work

The integration of communication and synchronization paradigms has been explored in many contexts. The SR language (Synchronizing Resources) developed by Andrews [1, Section 10.4] permits concurrent processes to communicate and synchronize using shared variables, semaphores, asynchronous message passing, RPC, and rendezvous, as does StarMod [5]. The unique aspects of our work are the integration of these paradigms within the framework of object-orientation, and the assertional verification of the implementation at the level of the code itself.

3 The CC++ Language

CC++ is a parallel object-oriented programming language based on C++. It makes use of single-assignment variables, atomic functions, and parallel composition to express and control concurrency. The following two CC++ constructs (used in the monitor and asynchronous channel libraries) control synchronization and interleaving:

`sync:` a single-assignment object.

`atomic:` a function whose statements are not interleaved with statements that are not part of the function. Atomic functions must not suspend and must terminate.

The example programs use the following constructs contained in the language for expressing parallelism:

`par:` a block of statements that are executed in parallel. The block terminates only when all its statements have terminated.

`parfor:` a loop whose iterations are executed in parallel. The loop terminates only when all its iterations have terminated.

For a complete description of these constructs, refer to [3] and [8].

4 Monitor Library

4.1 Specification

A monitor [2, 6] is a collection of data and functions that manipulate this data. A monitor provides synchronization between concurrent processes in two ways: only one process at a time is permitted to be inside a monitor executing any of its member functions, and processes executing inside the monitor can synchronize by means of special variables called *condition variables*. The operation `wait()` on a condition variable c causes the executing process to be suspended and placed in a pool of blocked processes associated with c. The `signal()` operation on condition variable c causes a blocked process associated with c to become ready to execute. We choose here to allow the signaling process to continue executing inside the monitor.

Monitors guarantee mutual exclusion. Let `#enter` be the number of times processes have entered the monitor (and begun executing) and let `#leave` be the number of times they have left the monitor (because the function being executed either terminated or performed a `wait()`). The first safety condition is then:

$$(\#\texttt{enter} = \#\texttt{leave}) \vee (\#\texttt{enter} = \#\texttt{leave} + 1) \quad (1)$$

If every member function of a monitor either terminates or executes a wait, then every function that becomes ready to execute is eventually allowed to enter. Unfortunately, programmers can write functions that do not terminate. Hence, only a much weaker property can be stated. Let `#became_ready` be the number of processes that became ready to execute (either by calling a monitor function, or by being reawakened after a `wait()`). Then:

$$\#\texttt{enter} = \min(\#\texttt{became\_ready}, \#\texttt{leave} + 1) \quad (2)$$

We distinguish between the two cases for a `signal()`: either there is at least one process waiting on the signaled condition variable, or there is none. In the former case, the `signal()` is designated as *heard* since it causes a process to become ready, and in the latter case it is designated as *ignored* since it has no effect. If the number of waits exceeds the number of heard signals, then the next signal will be heard. Initially there are no waiting processes. Thus, if $\Sigma = \{\mathbf{w}, \mathbf{s}_i, \mathbf{s}_h\}$ is the alphabet representing the operations wait, ignored signal, and heard signal respectively, and $V^c \in \Sigma^*$ is the string of operations that have been performed on condition variable c, then:

$$\exists s : s \in \mathcal{L}(G) : \quad V^c \leq s \quad (3)$$

where $w \leq z$ for two strings w and z denotes that w is a prefix of z and where G is the context-free grammar defined by the start symbol S, the set Σ of terminals, and the production rules:

$$\begin{aligned} S &\rightarrow \mathbf{s}_i S \mid BS \mid \epsilon \\ B &\rightarrow B\mathbf{w}B\mathbf{s}_h \mid \epsilon \end{aligned}$$

4.2 Design and Implementation

4.2.1 Mutual Exclusion

The mutual exclusion guaranteed by a monitor (equation 1) is provided by two member functions, `enter()` and `leave()`. The `enter()` function must be placed at the beginning of every user-defined member function, and the `leave()` function must be placed at the end of every user-defined member function.

The monitor maintains a private queue, `Ready`, of processes that are ready to execute, but prevented from entering by the mutual exclusion requirement. This queue is simply a collection of pointers to undefined `sync` objects. A process P suspends itself by creating a new undefined `sync` object (Figure 1-a), appending a pointer to it on the `Ready` queue, and then attempting to read the contents of the `sync` object (Figure 1-b). Eventually, this pointer comes to the head of the `Ready` queue (Figure 1-c) and is dequeued by an exiting process, Q (Figure 1-d). Process Q defines the `sync` object, awakening P (Figure 1-e). P then deallocates the `sync` object on which it was suspended and enters the monitor (Figure 1-f).

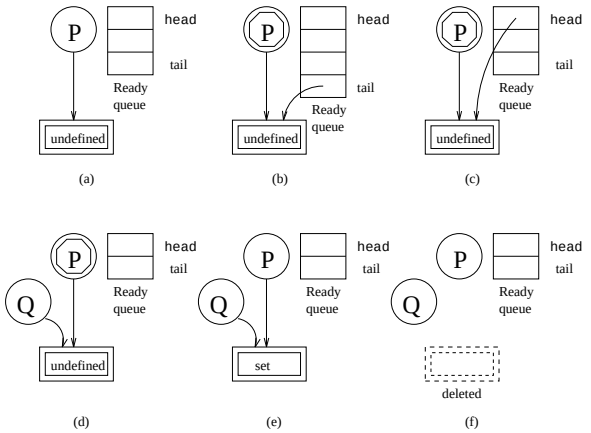

Figure 1: Monitor Ready Queue

4.2.2 Monitor as a Base Class

The concepts of data and function encapsulation of a monitor are consistent with those of C++ classes. In addition to the constructs that all monitors share (*e.g.* a definition of the condition variable type), a monitor must contain the particular member functions specific to its applications. Thus, a particular instantiation of a monitor will be a superset of the fundamental data type and function members that all monitors share. This functionality can be achieved by implementing the `Monitor` class as a base class.

4.2.3 Using the Monitor Library

An application that uses this library to implement a particular monitor must obey the following rules:

1. The user-defined monitor must be derived from the library monitor.
2. All member data must be private.
3. Every member function must begin with a call to `enter()`, and must terminate with a call to `leave()`. Thus, the outer block of a monitor member function must be a sequential block, and all member functions must be `void` functions.

The class declaration for the `Monitor` class is:

```
class Monitor {
  private:
    Boolean busy;
    Queue<sync int> Ready;
    atomic void check_busy(sync int *);

  protected:
    Monitor(void);
    void enter(void);
    atomic void leave(void);
    class Condition
      { public: Queue<sync int> Waiting; };
    void wait(Condition&);
    void signal(Condition&);
};
```

4.3 Verification

4.3.1 State

The state of a monitor is defined by:

1. Whether or not a process is inside the monitor. The Boolean flag `busy` is `TRUE` exactly when a process is executing inside.

$$\texttt{busy} \Leftrightarrow \#\texttt{enter} = \#\texttt{leave} + 1 \quad (4)$$

$$\neg\texttt{busy} \Leftrightarrow \#\texttt{enter} = \#\texttt{leave} \quad (5)$$

2. The queue (`Ready`) of processes that are ready to execute inside the monitor. The size of this queue is the difference between the number of processes that became ready and the number that entered the monitor.

$$\#\texttt{enter} \leq \#\texttt{became\_ready} \quad (6)$$

3. The queues of waiting processes associated with each condition variable. Let Q_k^c be the number of processes waiting on condition variable c after k operations on that variable. Initially, the wait queues are empty, so $Q_0^c = 0$. Since each `wait(`c`)`

operation appends a process and each `signal(`c`)` operation that is heard removes one, we have:

$$\forall k \quad : |V^c| \geq k \geq 0 : \quad 0 \leq Q^c_k \tag{7}$$

$$\forall k \quad : |V^c| \geq k \geq 1 : \quad Q^c_k = \left|V^c_{0..k-1} \restriction \{\mathbf{w}\}\right| - \left|V^c_{0..k-1} \restriction \{\mathbf{s}_h\}\right| \tag{8}$$

where $\restriction$ is the binary operator that projects a string and a set of literals into a string consisting only of those literals in the set.

Since `signal()`'s on empty queues are always ignored and `signal()`'s on non-empty queues are always heard, we have:

$$\forall k \quad : |V^c| > k \geq 0 : \quad Q^c_k = 0 \Rightarrow V^c_k \neq \mathbf{s}_h \tag{9}$$

$$\forall k \quad : |V^c| > k \geq 0 : \quad Q^c_k > 0 \Rightarrow V^c_k \neq \mathbf{s}_i \tag{10}$$

4.3.2 Property

Maximality of Progress As many processes as possible will be allowed to enter the monitor, subject to the constraints of mutual exclusion and of the number that became ready.

$$\texttt{busy} \vee (\#\texttt{enter} = \#\texttt{became\_ready}) \tag{11}$$

4.3.3 Correctness

By annotating the implementation with assertions and ghost variables, the assertions (4)-(11) can be shown to be invariantly true [7]. Note that the assertions need not hold inside of atomic actions, but only at the beginning and end of such actions. As an example, we include the annotation of the `leave()` member function.

```
atomic void Monitor::leave(void)    {

  // ASSERT:
  // busy
  // #enter = #leave+1
  // (4) - (11)

  // #leave++

  // ASSERT:
  // busy
  // #leave = #enter
  // (6) - (11)

  if (Ready.isempty())

    // ASSERT:
    // busy
    // #leave = #enter
    // #enter = #became_ready
    // (6) - (11)

    busy = FALSE;

    // ASSERT:
    // ¬ busy
    // #leave = #became_ready
    // (4) - (11)

  else

    // ASSERT:
    // busy
    // #leave = #enter
    // #enter < #became_ready
    // (6) - (11)

    *(Ready.dequeue()) = SET;
    // #enter++

    // ASSERT:
    // busy
    // #leave < #became_ready
    // (4) - (11)
}
```

Now the conjunction of the specifications (1)-(3) follows from the conjunction of these invariants. For example, (2) can be shown to follow from (4), (5), (6), and (11).

$$\begin{aligned}
& \texttt{TRUE} \\
\Leftrightarrow \quad & \{ \text{ by 4 and 11 } \} \\
& (\#\texttt{enter} = \#\texttt{became_ready}) \vee (\#\texttt{enter} = \#\texttt{leave} + 1) \\
\Leftrightarrow \quad & \{ \text{ property of min, and by 4, 5, and 6 } \} \\
& (\#\texttt{enter} \geq \min(\#\texttt{became_ready}, \#\texttt{leave} + 1)) \wedge \\
& (\#\texttt{enter} \leq \#\texttt{became_ready}) \wedge (\#\texttt{enter} \leq \#\texttt{leave} + 1) \\
\Leftrightarrow \quad & \{ \text{ property of min } \} \\
& (\#\texttt{enter} \geq \min(\#\texttt{became_ready}, \#\texttt{leave} + 1)) \wedge \\
& (\#\texttt{enter} \leq \min(\#\texttt{became_ready}, \#\texttt{leave} + 1)) \\
\Leftrightarrow \quad & \{ \text{ antisymmetry of } \leq \} \\
& \#\texttt{enter} = \min(\#\texttt{became_ready}, \#\texttt{leave} + 1) \quad \square
\end{aligned}$$

Similarly, (1) follows directly from (4) and (5), and (3) follows from (7) - (10) [7].

4.4 Example Program

Problem Description Producer processes deposit messages into a finite buffer, and consumer processes remove them. A producer may deposit a message if there is at least one empty slot. A consumer may remove a message if there is at least one full slot. Deposits and removals must be mutually exclusive to preserve the integrity of the buffer.

```
class BoundedBuffer : private Monitor {
  private:
    int size;
    char *Buf;
    int nextin, nextout, fullcnt;
    Condition notempty, notfull;
```

```
  public:
    BoundedBuffer(int n) {
      size = n;
      Buf = new char[size];
      fullcnt = 0;
      nextin = 0;
      nextout = 0;
    }

    void deposit (char data) {
      enter();
      while (fullcnt == size) wait(notfull);
      Buf[nextin] = data;
      nextin = (nextin+1)%size;
      fullcnt = fullcnt+1;
      signal(notempty);
      leave();
    }

    void remove (char &data) {
      enter();
      while (fullcnt == 0) wait(notempty);
      data = Buf[nextout];
      nextout = (nextout+1)%size;
      fullcnt = fullcnt-1;
      signal(notfull);
      leave();
    }
};
```

Discussion The usual C++ mechanism of inheritance is used to create a monitor. Each monitor function begins with a call to `enter()` and ends with a call to `leave()`. Data members are private and are initialized in the usual manner by the constructor. A reference parameter is used to return a value from the `remove()` monitor function, because of the void function restriction.

5 Asynchronous Channel Library

5.1 Specification

An asynchronous channel is a first-in-first-out message-passing buffer. Two operations are defined on such a channel. The `nonblockingSend()` operation places a message in the buffer. It never suspends. The `blockingReceive()` operation removes the next message from the buffer. If there is no message to be removed, this operation suspends until there is such a message. The channel can be used by arbitrary and varying numbers of producers and consumers.

Since the `blockingReceive()` operation can suspend, it is not atomic, and we distinguish between the initiation and the termination of this operation. Let `iR` be the number of `blockingReceive()` operations that have been *initiated*, and let `cR` be the number of `blockingReceive()` operations that have *completed*. Since the `nonblockingSend()` operation is atomic, no such distinction is required, and we define `cS` to be the number of completed `nonblockingSend()` operations.

The first safety condition is that as many `blockingReceive()` operations as possible have completed, subject to the constraints of the number of initiated `blockingReceive()` operations and the number of completed `nonblockingSend()` operations.

$$\texttt{cR} = \min(\texttt{iR}, \texttt{cS}) \tag{12}$$

Another safety condition is that the channel delivers the messages in FIFO order. Let the sequence of messages sent on the channel be $\texttt{s}_{(0)}, \texttt{s}_{(1)}, ..., \texttt{s}_{(\texttt{cS}-1)}$, and let the sequence of messages received on the channel be $\texttt{r}_{(0)}, \texttt{r}_{(1)}, ..., \texttt{r}_{(\texttt{cR}-1)}$. The k^{th} message received is the k^{th} message sent.

$$\forall k : 0 \leq k < \texttt{cR} : \texttt{r}_{(k)} = \texttt{s}_{(k)} \tag{13}$$

5.2 Design and Implementation

5.2.1 Asynchronous Channel as a Class

An asynchronous channel is implemented as a CC++ class. It stores the sent messages that have not yet been received in a private queue, `Undelivered`. The class has two public member functions: `nonblockingSend()` and `blockingReceive()`. The `nonblockingSend()` member function is atomic so that it can safely manipulate the `Undelivered` queue. The class also contains a queue, `EmptySlots`, to keep track of the suspended `blockingReceive()` operations.

The channel implementation is general enough to permit any object, including user-defined structures, to be sent as a message. This generality is achieved (without sacrificing the benefits of type-checking) by using templates.

5.2.2 Messages and Slots

Producers send messages on the channel and consumers give empty *slots* to the channel. A consumer waits for the slot to become full, and then processes the message in the slot. Slots are implemented as `sync` pointers. An empty slot is an undefined `sync` pointer. A full slot is a `sync` pointer that has been defined. The pointer points to the message contained in the slot.

A `blockingReceive()` operation begins by creating a slot for a message. If there is an undelivered message, the slot is filled and the operation terminates. If there is no such message, the slot is added to a queue of such slots, the `EmptySlots` queue. The `blockingReceive()` operation then suspends on the contents of this slot.

A `nonblockingSend()` operation begins by making a copy of the message being sent. If there are any suspended `blockingReceive()` operations, the first element of `EmptySlots` is dequeued and the `sync` pointer of the slot is defined to point to the message copy. If there are none, the message copy is appended to a queue of undelivered messages, `Undelivered`.

The class declaration for the asynchronous channel class is:

```
template <class Message>
class AChannel {
  private:
    Queue <Message * sync> EmptySlots;
    Queue <Message> Undelivered;
    atomic void give_slot (Message * sync *);

  public:
    atomic void nonblockingSend (const Message &);
    Message * blockingReceive(void);
};
```

5.3 Verification

5.3.1 State

The state of an asynchronous channel is given by:

1. A queue of empty slots called `EmptySlots`. The number of elements in this queue, `#EmptySlots`, is the number of `blockingReceive()` operations that have been initiated, but have not yet completed.

$$\#\texttt{EmptySlots} = \texttt{iR} - \texttt{cR} \tag{14}$$

   Empty slots are placed in this queue in order.

$$\forall j : 0 \leq j < \#\texttt{EmptySlots} : \texttt{EmptySlots}[j] = \texttt{r}_{(j+\texttt{cR})} \tag{15}$$

   where $\texttt{EmptySlots}[j]$ is the $(j+1)^{th}$ element of `EmptySlots`.

2. A queue of undelivered messages called `Undelivered`. Let the number of elements in this queue be `#Undelivered`. Since each completed `nonblockingSend()` either adds a message to this queue or allows a `blockingReceive()` operation to complete, we have:

$$\#\texttt{Undelivered} = \texttt{cS} - \texttt{cR} \tag{16}$$

   Undelivered messages are placed in this queue in order.

$$\forall j : 0 \leq j < \#\texttt{Undelivered} : \texttt{Undelivered}[j] = \texttt{s}_{(j+\texttt{cR})} \tag{17}$$

   where $\texttt{Undelivered}[j]$ is the $(j+1)^{th}$ element of `Undelivered`.

5.3.2 Properties

Boundedness Requirement. The number of `blockingReceive()` operations that can complete is bounded by the number of `nonblockingSend()` operations that have completed.

$$\texttt{cR} \leq \texttt{cS} \tag{18}$$

The Set of Suspended Processes is Minimal. A process can be suspended only if its completion would violate the safety condition given by (12).

$$(\#\texttt{EmptySlots} = 0) \vee (\#\texttt{Undelivered} = 0) \tag{19}$$

Ordering Requirement. The value returned by the $(k+1)^{th}$ `blockingReceive()` points to the message contained in the $(k+1)^{th}$ slot. Let $\texttt{slot}_{(k)}$ be the message contained in the $(k+1)^{th}$ slot.

$$\forall k : 0 \leq k < \texttt{cR} : \texttt{r}_{(k)} = \texttt{slot}_{(k)} \tag{20}$$

The $(k+1)^{th}$ message sent is put in the $(k+1)^{th}$ slot.

$$\forall k : 0 \leq k < \texttt{cR} : \texttt{s}_{(k)} = \texttt{slot}_{(k)} \tag{21}$$

5.3.3 Correctness

By annotating the implementation with assertions and ghost variables, the assertions (14)-(21) can be shown to be invariantly true. The conjunction of the specifications (12) and (13) follows from the conjunction of these invariants. [7]

5.4 Example Program

Problem Description Each process in a mesh begins with an initial value. At each iteration, every process calculates its new value as a weighted average of its old value and the values of its neighbors. The problem is to find the final values after a fixed number of iterations.

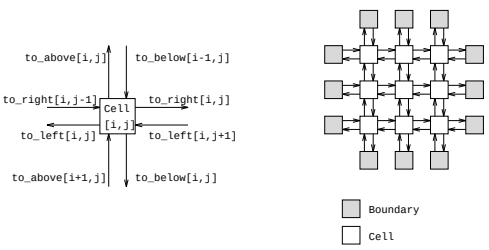

Figure 2: Mesh Structure for Dirichlet's Problem

```
const int M = 7;                                    //size of mesh
const int I = 100;                                  //number of iterations
typedef AChannel<float> chan;

float Cell (chan &fr_W, chan &fr_E, chan &fr_N, chan &fr_S,
            chan &to_W, chan &to_E, chan &to_N, chan &to_S,
            float value) {
  float *W_val, *E_val, *N_val, *S_val;
  for (int time=0; time<I; time++) {
    to_W.nonblockingSend(value);
    to_E.nonblockingSend(value);
    to_N.nonblockingSend(value);
    to_S.nonblockingSend(value);
    W_val = fr_W.blockingReceive();
    E_val = fr_E.blockingReceive();
    N_val = fr_N.blockingReceive();
    S_val = fr_S.blockingReceive();
    value = (4*value+*W_val+*E_val+*N_val+*S_val)/8.0;
    delete W_val;
    delete E_val;
    delete N_val;
    delete S_val;
  }
  return value;
}

float Boundary (chan &f_int, chan &t_int, float value) {
  float * in_val;
  for (int time=0; time<I; time++) {
    t_int.nonblockingSend(value);
    in_val = f_int.blockingReceive();
    delete in_val;
  }
  return value;
}

void main ()
{
  chan to_E[M][M], to_W[M][M], to_N[M][M], to_S[M][M];
 //to_x[i][j] is Cell ij's output channel in direction x
  float Final[M][M];

  parfor (int i=1; i<M-1; i++) {
    par {
      Boundary(to_N[1][i], to_S[0][i],i);
      Boundary(to_E[i][M-2], to_W[i][M-1], i+M);
      Boundary(to_S[M-2][i], to_N[M-1][i], i+M*2);
      Boundary(to_W[i][1], to_E[i][0], i+M*3);
      parfor (int j=1; j<M-1; j++)
        Final[i][j] = Cell(to_E[i][j-1], to_W[i][j+1], to_S[i-1][j],
                           to_N[i+1][j], to_W[i][j], to_E[i][j],
                           to_N[i][j], to_S[i][j], 0);
    }  /*par*/
  }  /*parfor*/
}
```

Discussion Channels are passed as reference parameters; otherwise a local copy of the channel would be made. Sends and receives performed on a local copy would not affect the original channel.

6 Conclusion

We have presented a library integration of two imperative communication and synchronization paradigms in CC++: monitors and asynchronous channels. Each construct has been specified formally, implemented, and rigorously verified. Example programs have been given to illustrate the ease with which these concepts can be used in CC++. A similar approach has also been used to integrate semaphores and ported channels in CC++ [7].

It is interesting to note the brevity and simplicity of the proofs of correctness for these libraries. Only predicate logic was required in each case, since the specifications were given entirely in terms of safety properties. This was possible because of the concept of atomicity in CC++.

Object-orientation plays an important, facilitating role in the integration of these paradigms. Inheritance provides a natural mechanism for declaring and defining particular monitor instances. Generic classes permit the asynchronous channel library to be strongly typed, providing a more robust interface.

Our libraries are currently being used by developers in the fields of matrix algorithms and grid-based solvers. We are encouraged by these preliminary applications that these verified libraries will be of great value to programmers.

References

[1] Gregory R. Andrews. *Concurrent Programming: Principles and Practice.* Benjamin/Cummings, Redwood City, CA, 1991.

[2] Per Brinch Hansen. *Operating System Principles.* Prentice Hall, 1973.

[3] P. A. Carlin, K. M. Chandy, and C. Kesselman. The Compositional C++ language definition. Technical Report CS-TR-92-02, Computer Science Department, California Institute of Technology, 1992.

[4] K. Mani Chandy and Stephen Taylor. *An Introduction to Parallel Programming.* Jones and Bartlett Publishers, Boston, MA, 1992.

[5] R. P. Cook. Starmod— a language for distributed programming. *IEEE Transactions on Software Engineering SE-6*, 6:563–571, Nov 1980.

[6] C.A.R. Hoare. Monitors: An operating system structuring concept. *Comm.ACM*, 17(10):549–557, Oct 1974.

[7] P. A. G. Sivilotti. A verified integration of imperative parallel programming paradigms in an object-oriented language. Technical Report CS-TR-93-21, Computer Science Dept., California Institute of Technology, 1993.

[8] P. A. G. Sivilotti and P. A. Carlin. A tutorial for CC++. Technical Report CS-TR-94-02, Computer Science Dept., California Institute of Technology, 1994.